Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Aprendizaje por refuerzo en juego de mesa cooperativo mediante Unity ML-Agents

Autor: Óscar Arroyo Pastor

Tutor/es: Adrián Domínguez Díaz

2020

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Aprendizaje por refuerzo en juego de mesa cooperativo mediante Unity ML-Agents

Autor: Óscar Arroyo Pastor

Tutor/es: Adrián Domínguez Díaz

TRIBUNAL:

Presidente: León Atilano González Sotos

Vocal 1º: Luis de Marcos Ortega

Vocal 2º: Adrián Domínguez Díaz

FECHA: 28 de Setiembre de 2020

Índice

Resumen

En este proyecto se pretende realizar una investigación y aprendizaje sobre el motor de videojuegos Unity y las librerías que comprenden el ámbito del aprendizaje automático pertenecientes al motor.

Como aplicación de estas librerías, se diseñará e implementará una adaptación sencilla del juego de mesa cooperativo “Pandemic” al motor Unity. Y sobre este, se investigará el diseño y desarrollo de un jugador controlado mediante la inteligencia artificial que ayude a lo largo de la sesión de juego al resto de jugadores.

Abstract

This project aims to perform a research and study about the Unity video game engine and the libraries that cover the area of machine learning related to the engine.

As an application of these libraries, a simple adaptation of the cooperative board game "Pandemic" to the Unity engine will be designed and implemented. On this, it will be investigate the design and development of a player controlled by the artificial intelligence that will help the rest of the players along the session.

Palabras Clave

Unity, ML-Agents, Inteligencia Artificial, Aprendizaje por refuerzo, Pandemic.

Resumen Extendido

En este proyecto final de grado se pretende realizar una aproximación al manejo del motor de videojuegos Unity, así como investigar las librerías existentes para este pertenecientes al campo de la inteligencia artificial, en concreto al aprendizaje automático.

Una vez se hayan adquirido unos conocimientos mínimos sobre el funcionamiento de Unity y el desarrollo de juegos en este a través de varios ejemplos y mini proyectos sencillos, se realizará una búsqueda e instalación de las librerías y el resto de las herramientas necesarias para la introducción al aprendizaje automático.

Al igual que en los primeros pasos y mini proyectos estudiados en Unity, se realizará la misma tarea, aplicada al aprendizaje automático. Es decir, se realizarán varios ejemplos y mini proyectos para aprender en el uso de estas librerías con el objetivo de su posterior aplicación a nuestro proyecto.

Con la infraestructura ya montada y soltura en la tecnología, se diseñará y desarrollará una adaptación sencilla del juego de mesa “Pandemic”, sobre el que se implementará posteriormente el agente a diseñar. Para implementar el aprendizaje automático, se usará la técnica del aprendizaje por refuerzo con los algoritmos disponibles en las librerías y herramientas estudiadas e instaladas previamente.

Para alcanzar este cometido, se utilizarán unas reglas más sencillas a las existentes en el juego normal. Donde un jugador controlado por la máquina ejecute las acciones disponibles que se pueden realizar durante su turno en la partida, de forma óptima y más natural e inteligente posible.

Con este propósito, se dividirá el proyecto en varias fases de desarrollo. Empezando por desarrollar un agente capaz de moverse por los nodos que forman el tablero del juego. En la siguiente fase, teniendo ya un movimiento perfeccionado, se pasará a desarrollar e implementar otras de las acciones disponibles para el jugador, la curación de virus existentes en el tablero automáticamente cuando se visita un nodo. Por último, se tratará de desarrollar que el agente,

elija en su turno si moverse a alguno de los nodos, o curar en el nodo en el que se encuentra, tomando así decisiones tal y como lo haría un jugador humano.

En todas estas fases se aplicarán los diferentes algoritmos y métodos de aprendizaje de las librerías, así como diferentes mejoras y comparaciones entre los probados, dando constancia de los mejores resultados. La conclusión de estas fases consistirá en analizar cómo el agente se comporta en una partida normal del juego adaptado, y como podría interactuar con otros agentes entrenados o jugadores humanos.

Introducción

Planteamiento del Trabajo

En el presente trabajo se pretende aprender los conocimientos básicos del motor Unity, tanto en su apartado del desarrollo de videojuegos como en al ámbito del aprendizaje automático. Como muestra de este aprendizaje, se desarrollará e implementará un juego y un agente entrenado que pueda jugar.

Para ello, se ha usado la librería de Unity ML-Agents. Esta librería ha proporcionado los algoritmos necesarios para realizar el entrenamiento de un agente desarrollado y entrenado mediante el aprendizaje por refuerzo.

Por otro lado, se ha usado el juego de mesa "Pandemic" como juego a adaptar a Unity. Se trata de un juego de mesa cooperativo en la que los jugadores, a lo largo de un mapa mundial de ciudades tratadas como nodos, deben cooperar para curar y erradicar los distintos virus que aparecen en las ciudades a lo largo de la partida. Para ello disponen de distintas acciones que pueden realizar durante su turno.

Con estos conceptos, se ha adaptado el juego al motor Unity con unas reglas más sencillas que las del juego principal. Y, además, se ha diseñado, desarrollado e implementado un agente entrenado capaz de jugar al juego adaptado siguiendo las reglas.

A lo largo de la memoria, se profundizará en el uso de Unity en sí, con algunos mini proyectos y adaptando el juego de mesa. Se profundizará también en el uso de ML-Agents con ejemplos similares a los de Unity, y se combinarán todos estos conocimientos para desarrollar y explicar el aprendizaje automático aplicado al juego adaptado.

Punto de Partida

En este trabajo no se ha partido de ninguna base o modelo anterior para su desarrollo, por lo que se puede decir que el desarrollo de todo el trabajo ha partido desde cero. Sin embargo, se ha de aclarar que actualmente en Unity existen modelos, videojuegos y otros proyectos en los que uno o varios agentes entrenados desarrollan tareas propias de cada uno de los juegos. También, es escaso el desarrollo y aplicación en juegos de mesa adaptados usando la tecnología de Unity, dada la complejidad de los datos a recoger o la cooperación entre agentes.

En cuanto al propio juego a adaptar, existe una versión digital del mismo disponible para comprar. Se han utilizado las reglas y el propio juego de mesa para realizar su adaptación, y programar las acciones disponibles para el agente.

Estado del Arte

En estos momentos existen infinidad de aplicaciones de la IA entrenadas mediante el método de aprendizaje por refuerzo en muchos campos como en la simulación de vehículos o la robótica [1].

Con respecto al ámbito de los videojuegos, desde el comienzo de la creación de estos se ha pensado en el desarrollo de agentes controlados por la propia máquina. El principal problema del desarrollo de IA es el propio algoritmo. Es decir, cada juego precisaba de un algoritmo específico sobre el que la IA se basaba para jugar al propio juego. Y como es de esperar, esto supone un esfuerzo y tiempo muy elevado en el desarrollo de los propios agentes.

Sin embargo, gracias al aprendizaje por refuerzo se ha conseguido solventar en parte este tipo de problemas, desarrollando una serie de algoritmos “comunes” los cuales pueden usarse para el entrenamiento de las IAs en prácticamente cualquier videojuego.

Uno de los progresos más importantes alcanzados en la IA en juegos, fue en el año 2013 de la mano de la empresa DeepMind. El artículo “Playing Atari with Deep Reinforcement Learning” de la mano de Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al., comenta como utilizando el aprendizaje por refuerzo en combinación de un nuevo modelo de aprendizaje profundo, recibiendo solo información de los píxeles como datos de entrada es capaz de dominar varios

juegos de la consola Atari 2600 alcanzando un comportamiento en el juego muy similar al de un humano [2]. Además, se desarrolló un algoritmo (Deep Q-Networks o DQN) que permitió facilitar el entrenamiento de redes profundas a través del aprendizaje por refuerzo.

Años más tarde, en el año 2015, la misma empresa presentó un programa llamado AlphaGo que jugaba al clásico juego "Go". En el artículo "Mastering the Game of Go with Deep Neural Networks and Tree Search" escrito por David Silver, Aja Huang, Chris J. Maddison, et al., se habla de cómo "Go" ha sido uno de los juegos clásicos más complicados para la inteligencia artificial, debido a factores como el espacio de búsqueda o las posiciones y movimientos dentro del espacio de juego [3]. Se explica también cómo se ha ido desarrollando el programa "AlphaGo" hasta su versión final. Desde el aprendizaje supervisado para predecir los movimientos de un experto en el juego, hasta poco a poco ir introduciendo elementos característicos del aprendizaje por refuerzo. Como por ejemplo las funciones de valor o las políticas de la red.

Con todos estos elementos se aplicó un nuevo algoritmo de entrenamiento con la técnica del aprendizaje por refuerzo en combinación del algoritmo de Monte Carlo. Al año siguiente, dicho programa venció al actual campeón mundial del juego por cinco partidas a cero. Convirtiéndose en la primera vez que un programa de ordenador derrotó a un jugador profesional humano en el juego completo "Go".

Tratando un poco más la temática de los juegos mesa, artículos como "Reinforcement Learning of strategies for Settlers of Catan" de Michael Pfeiffer estudia cómo aplicar métodos del aprendizaje automático en un juego de mesa complejo como es "Los colonos de Catán" [4]. Explica los problemas encontrados introduciendo el aprendizaje por refuerzo para el entrenamiento, y que debido a la cantidad de acciones y complejidad de estados del juego es necesario realizar otros acercamientos. Es aquí donde introduce la heurística en combinación con el aprendizaje por refuerzo para tratar de resolver las tareas más complicadas.

Abordando el aprendizaje por refuerzo en videojuegos con el uso de Unity, existen varios artículos interesantes como pueden ser los siguientes.

Samuel Arzt y Gerhard Mitterlechner, et al. De la Universidad de Ciencias Aplicadas de Salzburg, escribieron un artículo donde se explicaba y se aplicaban varios métodos del aprendizaje por refuerzo profundo como DQN de DeepMind o A2C, para entrenar a un agente en un videojuego

moderno con un entorno 3D [5]. Usando la misma técnica que el entrenamiento para juegos Atari, es decir, con unos datos de entrada para el modelo de la red neuronal de solamente la información de los pixeles de la pantalla. Donde su trabajo se centra en el impacto que pueda tener estas modificaciones del entorno en el rendimiento del aprendizaje. Siendo algunas modificaciones el entorno 3D, el control del agente a nivel de físicas de videojuego, etc. Todo ello con la tecnología que proporciona Unity y la librería de ML-Agents.

Otro artículo de características similares al presente trabajo es "Implementing Artificial Intelligence Agent Within Connect 4 Using Unity3d and Machine Learning Concepts" de Nirmal Baby y Bhargavi Goswami, en el se explica cómo se ha adaptado el famoso juego de mesa "Conecta 4" al motor de videojuegos Unity3D para que, posteriormente, puedan aplicarse conceptos del aprendizaje automático a través de la librería disponible ML-Agents [6]. También explican cómo se han enfrentado a los problemas y dificultades encontrados a lo largo de la implementación y algunos resultados obtenidos una vez los problemas han sido solventados.

Objetivos

El objetivo de este trabajo consiste en aprender el uso de la tecnología de Unity para desarrollar y aplicar así un agente entrenado que ayudará al jugador a lo largo de la partida, pudiendo extender la duración de esta.

Para ello se tomará como base el juego de mesa "Pandemic". Una vez adaptado al entorno de Unity, se trabajará sobre él para que en combinación de la librería ML-Agents se diseñe, desarrolle y aplique el aprendizaje automático para entrenar un agente.

Como consecuencia, se profundizará en el uso de Unity como motor de videojuegos y más concretamente en su apartado del aprendizaje automático o Machine Learning. Usando una de las disciplinas de este, como es el aprendizaje por refuerzo, para alcanzar los objetivos planteados.

Conceptos Teóricos

Inteligencia Artificial

Se conoce como inteligencia artificial a un campo del estudio de las ciencias de la computación que busca explicar y emular comportamiento inteligente en términos de procesos computacionales [7].

Una de las formas conocidas de conseguir emular este comportamiento es con el uso de las redes neuronales. Esto es, un modelo computacional que se asemeja vagamente en las redes neuronales del cerebro humano a través del cómputo de unas fórmulas y algoritmos específicos.

Para que estas redes puedan servir de inteligencia para un agente máquina o computadora, tienen que pasar tres fases distintas. Una donde se diseñe la red, una fase entrenamiento y una vez pasada, una fase de prueba de la red. La manera de aprender de estas redes se denomina aprendizaje automático o Machine Learning (ML). Son un tipo de IA que aprenden a realizar una tarea concreta a través de la evaluación de datos, sin necesidad de estar programada para ello en un principio. Existen tres tipos de paradigmas de ML [8]:

- Aprendizaje supervisado. Se trata de un tipo de aprendizaje que se caracteriza cuando es posible para el programador proporcionar al modelo todos los datos de entrada etiquetados con su correspondiente salida para el entrenamiento. Es decir, el modelo posee un conjunto de datos claros y normalizados.
- Aprendizaje no supervisado. En este otro tipo, al modelo solo se le entregan datos de entrada sin estar etiquetados de forma explícita. Por lo que es el propio modelo el encargado de relacionar o estructurar los datos de entrada, siendo la estructura desconocida por el programador.
- Aprendizaje por refuerzo. Consiste en un tipo de aprendizaje con un sistema de recompensas y castigos que se aplican sobre el agente máquina o computadora para que resuelva un determinado problema por sí mismo. Más adelante se profundizará sobre este tipo, ya que es clave en el desarrollo del trabajo.

Un aspecto en común a los tres métodos de aprendizaje son las fases de entrenamiento y de prueba de la red neuronal. Aunque a bajo nivel cada uno de los paradigmas funcionan diferente por la manera de tratar los datos, generalizando, en los tres se involucra una fase de entrenamiento donde se construye y diseña un modelo en base al proceso de una serie de datos obtenidos. Y una fase de prueba donde se usa el modelo entrenado con unos datos completamente nuevos a los del entrenamiento [9].

Redes neuronales

Las redes neuronales usadas para el aprendizaje automático están compuestas de los siguientes componentes [7]:

- Neuronas. Son los componentes principal de la redes y está definido como una función matemática. Esta posee una o varias entradas y una sola salida. La entrada puede ser desde un simple dato externo a la información de la salida de otra neurona. En el caso de la salida puede ser la respuesta final de la red, o la entrada a otra neurona.
- Conexiones. Es la forma que hace posible la comunicación entre las diferentes neuronas. Una neurona puede tener múltiples conexiones.
- Peso. Cada una de las conexiones de la red está dotada de un peso que representa la importancia del dato que recibirá la neurona por esa vía.
- Función de propagación. Corresponde a la función matemática de la neurona que procesa la entrada con los pesos y establece la salida.

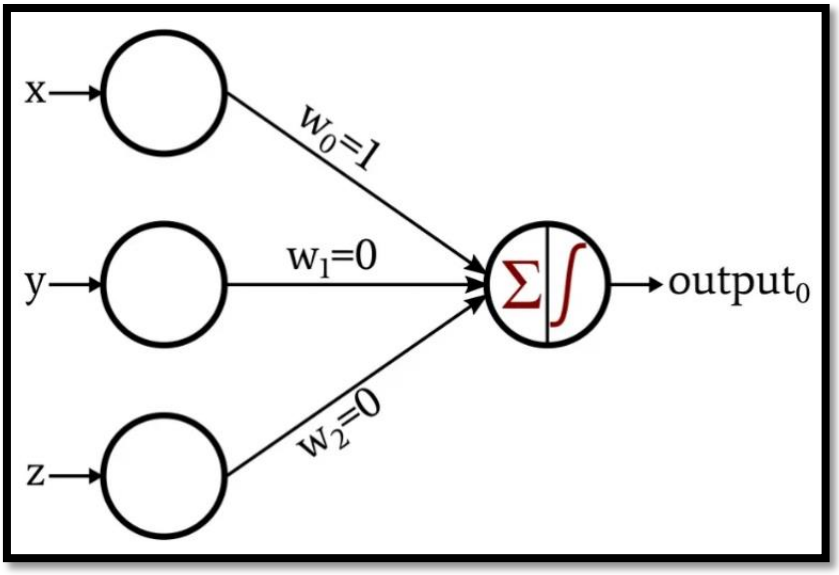


FIGURA 1. RED NEURONAL SIMPLE O PERCEPTRÓN [10].

Generalmente las redes neuronales se organizan en capas, unas enlazas con otras, lo que da lugar a distinguir entre dos tipos de redes neuronales:

- Redes neuronales artificiales o artificial neural networks (ANNs). Son las comúnmente conocidas como redes neuronales. Sistemas que mediante cómputo aprenden tareas específicas para resolver un problema.

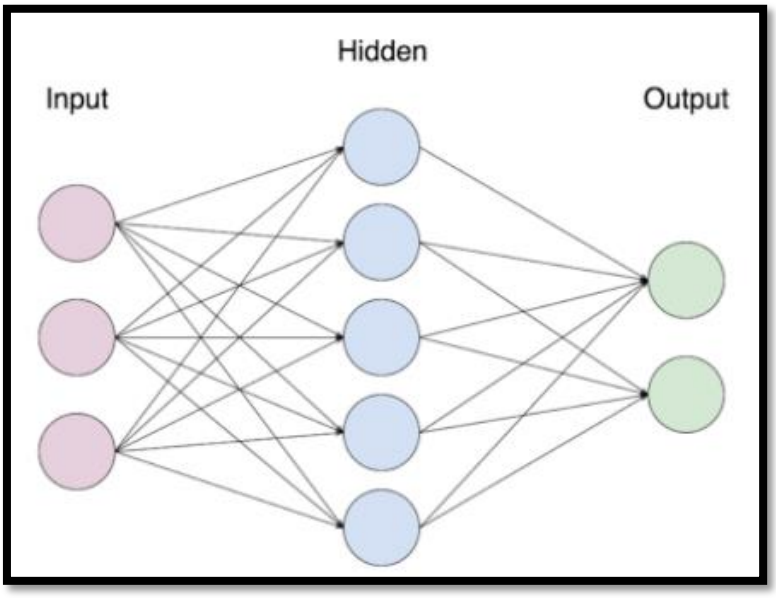


FIGURA 2. RED NEURONAL ARTIFICIAL SIMPLE [11].

- Redes neuronales profundas o deep neural networks (DNNs). Son redes neuronales artificiales con la diferencia de que poseen múltiples capas intermedias entra las capas de entrada y de salida de la red. Estas redes son capaces de transformar matemáticamente los datos de entrada en datos de salida, independientemente del tipo de conexión entre las neuronas. Cada transformación realizada se corresponde con una capa de la red [12].

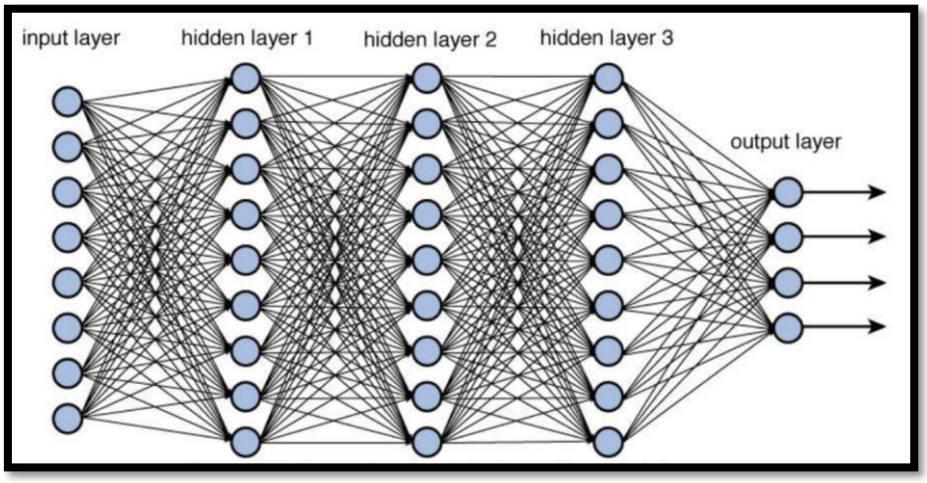


FIGURA 3.RED NEURONAL PROFUNDA [12].

Aprendizaje por refuerzo

El aprendizaje por refuerzo o Reinforcement Learning (RL) es un área del aprendizaje automático donde a una IA se le presentan un entorno y unas posibles acciones a tomar, que en función de la acción tomada dará una recompensa. A lo largo de una serie de iteraciones, la IA debe aprender por ella misma a maximizar dicha recompensa a través de la elección de las acciones. Gracias a este tipo de aprendizaje, la IA puede realizar tareas mucho más complejas que con otros métodos de aprendizaje [8], [13].

Dentro del aprendizaje por refuerzo, podemos encontrar cuatro elementos clave involucrados en el proceso de aprendizaje. Estos son: una política, una recompensa, una función de valor y un modelo de entorno [14].

- Política. Consiste en el comportamiento que un agente toma en cada iteración siguiendo los datos del estado del entorno en que está entrenando. Puede variar entre una lista

de datos simples, a funciones y datos complejos. Sin estos datos del entorno, sería imposible determinar el comportamiento y la capacidad para entrenar.

- Recompensa. Es una cantidad numérica que se le asigna al agente con la acción realizada en cada iteración. Esta cantidad puede ser tanto negativa, también llamada "castigo", como positiva, llamada recompensa. El objetivo del agente, como se ha comentado anteriormente, es conseguir la máxima recompensa. Es importante también remarcar la importancia de esta, ya que dicta los cambios en la política del agente de cara a futuras iteraciones en el entorno
- Función de valor. Siendo la recompensa la cantidad inmediata a cada acción tomada, la función de valor representa la recompensa total que un agente puede esperar a lo largo de varias iteraciones con el entorno. Es evidente la importancia de la función de valor, puesto que también dicta las acciones a escoger. Un agente no solo buscará una recompensa para una acción inmediata, si no que influenciará la acción que desencadene un estado en la que el valor total sea mayor.
- Modelo de entorno. En algunos sistemas de aprendizaje por refuerzo existen los modelos del entorno, esto es, una predicción de cómo es comportará el entorno en un estado posterior a la acción y estado actual. Son usados en la planificación, considerando posibles futuros estados a partir de situaciones anteriores.

Como es evidente, el aprendizaje por refuerzo depende en su mayoría del concepto de "estado", es decir, el conjunto que forman cada uno de los subelementos definidos en el párrafo anterior. La definición técnica de estado viene dada por el proceso de decisión de Markov.

Proceso de decisión de Markov

El proceso de decisión de Markov o Markov decision process (MDP) es la teoría matemática en la que está basado el problema del aprendizaje por refuerzo. En otras palabras, el MDP describe de una manera formal el entorno en el que se desarrolla el aprendizaje por refuerzo. Este entorno es completamente observable por el agente, lo que permite transformar el problema en un MDP. Definiéndose así una escena agente-entorno [14].

Esta escena tiene como objetivo especificar en el aprendizaje de cada iteración y convertirlo en conseguir una cierta meta. La parte que aprende y toma decisiones correspondería con el agente. Mientras que el "objeto" con el que interactúa y que compone todo lo que rodea al

agente, se denominaría entorno. Estos, como ya se ha mencionado, interactúan continuamente llevando a una situación de acción-reacción donde la reacción del entorno sería la recompensa. Esta interacción continua viene dada por una secuencia de pasos de tiempo discretos. Es decir, en cada instante de tiempo, o paso, el agente recibe la información del estado del entorno y decide que acción realizar. Un paso después, como respuesta a la acción escogida, se le otorga una recompensa numérica al agente y se le presenta el nuevo estado del entorno.

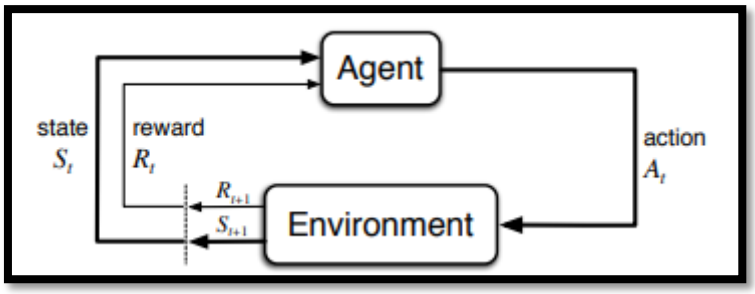


FIGURA 4. ESQUEMA PROCESO DE DECISIÓN DE MARKOV [14].

En el aprendizaje por refuerzo entonces la meta del agente es obtener una recompensa, preferiblemente, con un valor positivo. En cada paso este valor puede aumentar o disminuir, y el agente deberá buscar no la mejor recompensa para cada instante, si no maximizar la recompensa acumulada de cara a una recompensa total. Esta idea se denomina informalmente "hipótesis de recompensa".

El uso de una recompensa como respuesta a las acciones del agente para representar esta meta es una de las características más significativas del aprendizaje por refuerzo. Aunque parezca que puede ser una limitación, se ha probado en la práctica que fácilmente aplicable y flexible a cualquier tipo de problema.

Esta idea informal de recompensa puede expresarse de forma general como maximizar el retorno esperado, donde este retorno es una función de las secuencias de recompensas. El caso más simple de retorno de una acumulación de recompensas puede verse en la siguiente figura:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

FIGURA 5. FUNCIÓN DE RETORNO MÁS SIMPLE [14].

Donde "*G"* representa el retorno, *"R"* la recompensa, *"t"* el instante de tiempo o paso y *"T"* el paso final. Estas subsecuencias o pasos de tiempo se denominan formalmente episodios.

Prácticamente todos los algoritmos usados en el aprendizaje por refuerzo contienen una función de valor, esta determina una estimación de cuán buena es la decisión de un agente en una iteración del entorno. La manera de medir esta estimación es con la recompensa futura, o como se ha explicado en el párrafo anterior, la función de retorno esperada.

De acuerdo con esto, las funciones de valor se definen dependiendo de la manera de decidir del agente, denominadas políticas. Formalmente una política es un conjunto de los estados del entorno y de las probabilidades que existen de que el agente seleccione una acción sobre las demás. En los métodos usados en el aprendizaje por refuerzo la política de un agente es cambiante dependiendo de la experiencia adquirida en situaciones del entorno anteriores.

En resumen, una tarea en el aprendizaje por refuerzo implica encontrar una política que otorgue la mayor recompensa posible. Para los MDPs finitos, una política óptima se define como la política con el mayor o igual retorno esperado con respecto a las demás políticas posibles. Sin embargo, es posible que exista más de una política óptima. En consecuencia, una política óptima posee la función de valor óptima con respecto al resto.

Aprendizaje por refuerzo profundo

El aprendizaje por refuerzo profundo o Deep Reinforcement Learning (DRL) es una extensión del aprendizaje por refuerzo, donde este se combina con el uso de las redes neuronales de aprendizaje profundo o Deep Learning (DL). La diferencia consiste en el uso de redes con múltiples capas en el proceso de aprendizaje por refuerzo, es decir, las redes neuronales profundas (DNNs) descritas en apartados anteriores. Esto permite la resolución de problemas mucho más complejos [1], [15].

Desarrollo

Para el desarrollo del proyecto se utilizará la herramienta Unity y una de sus librerías disponibles llamada ML-Agents. El objetivo es desarrollar un agente que ayude en la partida curando en su turno los virus que vaya apareciendo. Puesto que es un proyecto que parte desde cero, el desarrollo se ha dividido en dos grandes partes. La división del trabajo sigue el siguiente esquema:

- Introducción. Donde se ha realizado el aprendizaje de la tecnología relacionada con el trabajo. Esta a su vez se ha subdividido en otras tres partes:
    - Motor Unity
    - Librería ML-Agents
    - Juego Pandemic
- Experimentos. Donde se ha investigado la aplicación de lo aprendido en la primera parte para poder desarrollar el aprendizaje automático de un agente. Además, también se ha subdividido en cuatro partes:
    - Movimiento del agente por camino más corto
    - Movimiento del agente por curación máxima automática
    - Decisión del agente entre movimiento o curación
    - Comparativa del algoritmo desarrollado con otros algoritmos.

Introducción

Unity

Unity es un motor para el desarrollo de videojuegos multiplataforma desarrollado por la empresa Unity Technologies. Entre las muchas funcionalidades que posee un motor de desarrollo de videojuegos, como el renderizado de objetos 2D y 3D, motor de físicas, animaciones, etc. Unity destaca por la sencillez y facilidad del motor, no lo solo a la hora del desarrollo de pequeños proyectos, si no por la cantidad de información y fácil acceso que posee el motor [16].

Además, también cabe destacar su apartado en el campo del aprendizaje automático, usado por muchas empresas tanto de simulación como de desarrollo de este ámbito.

ML-Agents

Información de la librería

Unity Machine Learning Agents o Unity ML-Agents es la librería de Unity que permite a los juegos desarrollados en la plataforma la simulación y el uso de entornos donde un agente puede entrenarse. Esta librería posee múltiples métodos del aprendizaje automático que permiten el entrenamiento del agente. Entre ellos, el aprendizaje por refuerzo.

El entrenamiento es posible gracias a la conexión de Unity y la librería con los algoritmos de entrenamiento a través de una API de Python. Todos los algoritmos que posee la librería están basados en la librería TensorFlow [9].

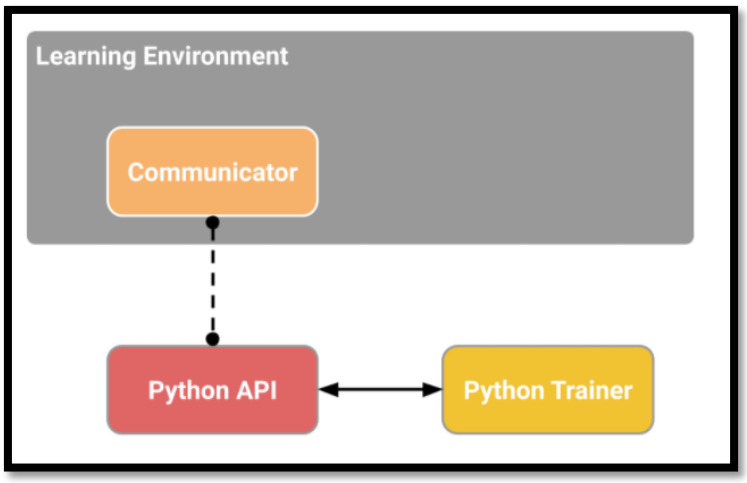


FIGURA 6. COMUNICACIÓN DE UNITY Y LA ML-AGENTS CON LA API DE PYTHON [9].

TensorFlow es una librería para realizar computación de datos a través del uso de grafos de flujo, que, en mayor o menor medida, es la representación de los modelos que forman el entrenamiento profundo. Gracias a esta librería se permite el entrenamiento y la prueba de redes neuronales con el uso tanto de la unidad central de procesamiento (CPU) como de la unidad de procesamiento gráfico (GPU).

Actualmente, la librería de ML-Agents posee tres algoritmos de entrenamiento de aprendizaje automático implementados.

- Behavioral Cloning (BC). Este método consiste en entrenar la política del agente imitando un conjunto de demostraciones previas.
- Generative Adversarial Imitation Learning (GAIL). Este método consiste en entrenar al agente dando una recompensa cuando su comportamiento es parecido al de un conjunto previo. A diferencia del anterior, en este método existe una segunda red neuronal llamada discriminador. En cada iteración del entrenamiento, el agente elige una opción dependiendo de la recompensa a obtener, y el discriminador examina la acción escogida fijándose si es similar al conjunto previo inicial dado o no. Cada vez volviéndose más crítico y acercando el comportamiento del agente al esperado.
- Deep Reinforcement Learning (DPL). Como ya se ha explicado, consiste en una extensión del aprendizaje por refuerzo con el uso de las DNNs. En el siguiente párrafo se extenderá sobre este algoritmo en la librería.

Cada uno de estos tres métodos pueden usarse tanto en conjunto como por separado.
Aunque existen muchos algoritmos en el aprendizaje por refuerzo profundo, la librería implementa los dos siguientes:

- Proximal Policy Optimization o PPO. Es el algoritmo más estable y aplicable de forma generalizada en contraste a la mayoría de los algoritmos de DRL. También es el algoritmo en el que se ha basado el entrenamiento del presente trabajo por lo que se profundizará más sobre él en apartados siguientes.
- Soft Actor-Critic o SAC. Se trata de un algoritmo sin política, por lo que el entrenamiento se basa en experiencias adquiridas en cualquier iteración. Estas experiencias son guardadas y escogidas de una forma aleatoria durante el entrenamiento. Esto supone una ventaja de eficiencia de tiempo sobre el resto de los algoritmos, pero al mismo tiempo también supone un mayor número de cambios en el modelo de la red.

Además de existir estos tres algoritmos de entrenamiento generales para cualquier tipo de entorno, también se implementan varios tipos de entrenamiento dependiendo del entorno con el que se va a entrenar al agente.

- Entrenamiento multi agente en entornos con juego individual. En este tipo de entorno uno a varios agentes entrena en un juego competitivo con la particularidad de que el

oponente puede ser otro agente que aprenda a la vez, o el mismo agente en versiones pasadas.

- Entrenamiento por Curriculum. Este método de entrenamiento hace que el agente entrene en un entorno más complejo a lo largo de todas las iteraciones totales con el entorno. Es decir, la idea es imitar el aprendizaje humano donde primero se aprende lo más básico, y a medida que el agente va adquiriendo una política óptima se va complicando el entorno. Esto permite al agente realizar entrenamientos de tareas mucho más complejas con mayor facilidad y de forma óptima que con otros algoritmos y métodos.
- Entrenamiento con parámetros aleatorios del entorno. Este tipo de entrenamiento está pensado para que un agente pueda adaptarse a situaciones del entorno que no han ocurrido durante el entrenamiento o se salen de lo habitual. Para ello, se establece uno o varios parámetros aleatorios en un objeto del entorno para que este cambie radicalmente en algún momento, permitiendo así al agente entrenar de manera más generalizada y adaptándose mejor a cambios del entorno.

Proximal Policy Optimization o PPO

Los métodos de política por gradiente son una serie de técnicas indispensables para el uso de redes neuronales profundas en materias como el control, los video juegos, etc. Pero estos métodos tienen una desventaja, la obtención de buenos resultados a través de ellos es complicada debido al tamaño del entrenamiento. Esto es, por ejemplo, siendo de un tamaño pequeño por lo que el progreso es lento. O por el contrario si el tamaño es grande desencadena un sobre entrenamiento o caídas en el rendimiento [17].

A lo largo del tiempo los investigadores han intentado solucionar el problema con acercamientos como el algoritmo TRPO (Trust Region Policy Optimization) o ACER (Actor Critic with Experience Replay), optimizando el tamaño de actualización de la política. Y aunque estos solucionan parte de los problemas citados anteriormente, ambos son más complejos de implementar que el PPO.

PPO otorga un equilibrio entre una implementación más sencilla que el resto de los algoritmos y con resultados en la actualización de la política durante el proceso de entrenamiento que no difiere mucho del resultado ideal. Este se basa en una función objetivo no presente en otros algoritmos que implementa una nueva forma de actualizar la política. Por lo que, al compararlo con el resto de los algoritmos, muestra el mejor rendimiento en tareas como las de control y se

asemeja a algunos algoritmos en otro tipos de problemas, pero siendo este mucho más sencillo en cuanto a implementación.

Diseño del entorno de entrenamiento

El entorno de entrenamiento en Unity está sujeto a dos clases de la librería ML-Agents, la clase "Academy" y la clase "Agent" [9]. La clase "Academy" es una clase de tipo Singleton que se encarga de manejar el entrenamiento del agente usando los objetos de la escena de Unity en cada una de las iteraciones. Durante el entrenamiento, la API de Python se comunica con esta clase mientras recoge los datos para la red y optimiza el modelo de red neuronal.

Todo entrenamiento de la librería sigue el siguiente bucle de procesos:

- Se reinicia el entorno llamando a la función "OnEnvironmentReset ()" de la clase "Academy".
- Se llama a la función "OnEpisodeBegin ()" de la clase "Agent".
- Se recoge la información que necesite el agente del entorno. Para ello, se utiliza la función "CollectObservations (VectorSensor sensor)" de la clase "Agent".
- El agente utiliza su política para decidir según el entorno y la información recogida la acción que va a realizar.
- Se llama a la función "OnActionReceived (float [] vectorAction)" de la clase "Agent" para ejecutar la acción elegida por la política.
- Se vuelve a llamar a la función "OnEpisodeBegin ()" del agente si este ha alcanzado su límite de pasos o ha finalizado una de las iteraciones del entrenamiento llamando a la función "EndEpisode ()" de su clase.

La clase "Agent" representa al agente que recoge la información del entorno y decide las acciones a tomar en consecuencia. Normalmente, esta clase se le asigna a un objeto de juego de Unity que haga del mismo agente. En el caso del presente proyecto, el jugador de la partida de "Pandemic".

El agente que extienda entonces de la clase "Agent" observa el entorno de entrenamiento y pasa toda la información obtenida a su política. Esta política corresponde a una clase "Policy" que realiza de forma trasparente al agente la lógica de decisión para pasarle a este la acción que

debe realizar. Cómo calcula esta clase cual es la mejor acción que realizar, depende de los "Behavior Parameters" o parámetros de entrenamiento que estén configurados en el agente. Estos parámetros se explicarán más adelante.

Como se ha explicado en el flujo de entrenamiento, la clase “Agent” implementa varias de las funciones necesarias para el entrenamiento, que pueden sobrescribirse para amoldarse al tipo de problema. Estas funciones pueden agruparse en los siguientes apartados:

- Recogida de información
- Ejecución de acciones
- Recompensa
- Toma de decisiones

La clase implementa varias maneras de recoger la información del entorno de entrenamiento dependiendo del tipo de dato. La información puede ser tanto un dato con valor numérico como información física. Para ello, se proporcionan tres tipos de observaciones o recolección de datos:

- Vectores de recolección. Utiliza la función “CollectObservations (VectorSensor sensor)” para almacenar la información. Cada “sensor” dentro de la función, llama al método “AddObservation” para añadir la información necesaria del entorno. Esta información puede variar desde la posición en la que se encuentra el agente, a los objetos que almacena un jugador en una mochila. Para el presente trabajo, la mayoría de las veces se ha recogido información del nodo donde se encuentra el jugador y de los nodos de destino o nodos con virus.
- Recolección visual. Almacena información en forma de imagen que después puede transformarse en un objeto 3D. Para ello se hace el uso del componente “CameraSensor” o “RenderTextureSensor”.
- Recolección raycast. Almacena información golpeada por los raycast o rayos en los diferentes objetos del entorno de entrenamiento. Esto permite a un agente en un entorno 3D saber los objetos que tiene alrededor. Simula una técnica similar a los sensores en un robot móvil.

En cuanto a la ejecución de las acciones, el agente invoca a la función “OnActionReceived (float [] vectorAction)” para realizar la acción o acciones decididas por la política del agente. El espacio del vector de acciones puede ser de dos tipos:

- Continuo. Este tipo de espacio proporciona un array de números decimales de un tamaño igual al del vector de la información recogida. Cada uno de los valores numéricos

representa un parámetro otorgado por la información recogida. El agente aprende que al usar dicho parámetro puede controlar el elemento del entorno correspondiente.

- Discreto. Este espacio proporciona un array con índices a las distintas acciones que se almacenan en "Branches" o ramas. Es decir, cada una de las ramas representa una serie de acciones que el agente puede realizar, y el valor de la rama un índice con la acción a realizar. Este tipo de espacio ha sido el usado en el actual trabajo, sobre todo en el último experimento.
    - Si el espacio a utilizar es de tipo discreto, es posible utilizar la función "CollectDiscreteActionMasks (DiscreteActionMasker actionMasker)" antes de realizar las acciones. Esta función permite "enmascarar" las acciones que el agente no puede decidir realizar, es decir, permite determinar qué acciones del total del vector no están disponibles para la actual iteración del entrenamiento. Seleccionando la rama sobre la que trabajar y las acciones de ella que no pueden elegirse como opción.

Dentro de la función anterior también puede establecerse la recompensa que se le da al agente por realizar la acción escogida. Para ello se hace uso de las funciones "AddReward ()" o "SetReward ()" dependiendo si se quiere añadir valor a la recompensa total o establecer el valor a la recompensa dada respectivamente. Es importante que la recompensa a otorgar se mantenga entre los valores -1 y 1 para que el entrenamiento sea estable.

Por último, para terminar de diseñar el entorno de entrenamiento se deben especificar los parámetros de la política del agente para la toma de decisiones y determinar el momento en el que el agente debe tomar una decisión. El archivo "Behavior Parameters" que contiene los parámetros sigue la siguiente Figura 7 de manera general.

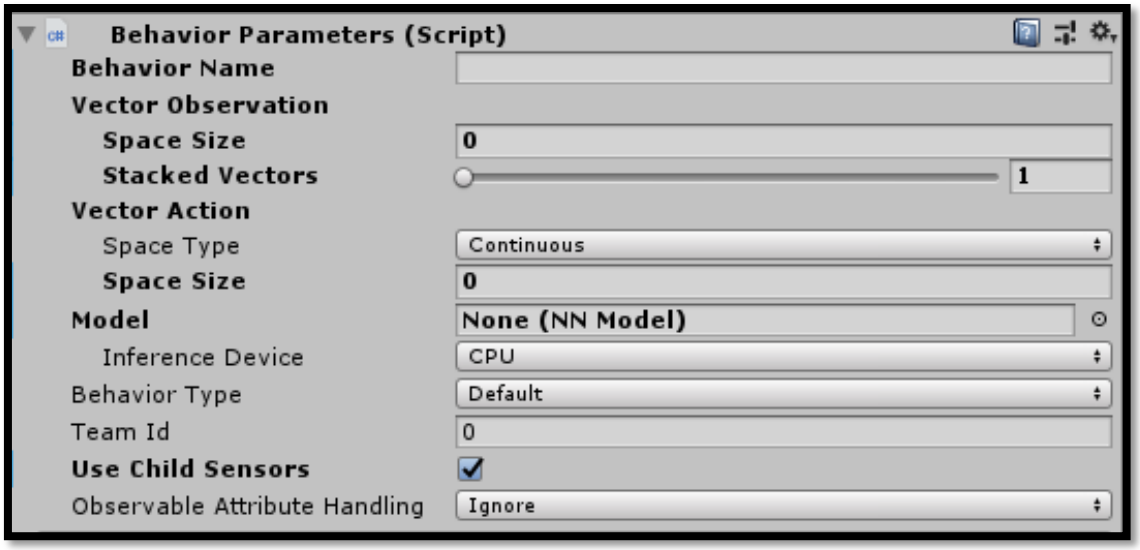


FIGURA 7. COMPONENTE "BEHAVIOR PARAMETERS" PARA LOS PARÁMETROS DE LA POLÍTICA DEL AGENTE.

- Behavior Name. Corresponde al nombre que se le da a la política del agente.
- Vector Observation
    - Space Size. Establece el tamaño del array que contiene la información del entorno recogida.
    - Stacked Vectors. Establece el número de vectores de información recogida anteriormente que se van a almacenar y utilizar para la decisión actual.
- Vector Action
    - Space Type. Establece el tipo de espacio de acción que se va a utilizar. Puede ser continuo o discreto como se ha citado anteriormente.
    - Space Size. En caso de especificar un tipo de espacio continuo, este parámetro establece el tamaño del vector de acción.
    - Branches. Si se establece un espacio de tipo discreto se usarán las ramas de acciones. Siendo "Branches Size" el número de ramas y "Branch N Size" el número de acciones que contiene la rama. Puede verse en la Figura 8.

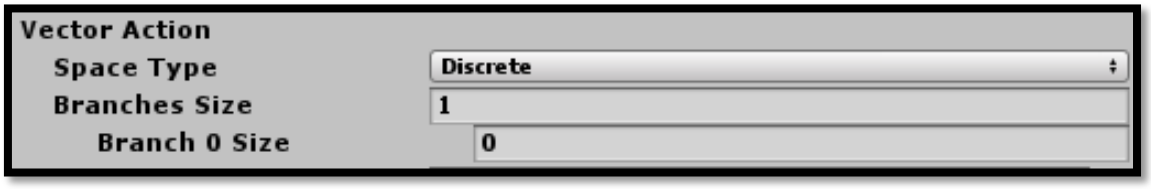


FIGURA 8. ESPACIO DE ACCIÓN DISCRETO EN EL COMPONENTE "BEHAVIOR PARAMETERS" DEL AGENTE.

- Model. Se le asigna un modelo de red neuronal en caso de tener una. Una vez entrenada se establece en esta opción el modelo a usar.
    - Inference Device. Establece el componente el ordenador que va a procesar el modelo de red al tener uno establecido. Puede procesarse mediante CPU o GPU.
- Behavior Type. Establece el tipo de comportamiento del agente en el entrenamiento. Puede seleccionarse entre tres opciones.
    - Default. Indica que el agente realizará un entrenamiento si al lanzar Unity está a la espera la API de Python con la que se conecta la librería. Si no, realizará inferencia.
    - Heuristic Only. El agente seguirá las órdenes programadas en la función "Heuristic ()" de la clase "Agent". Esta puede sobrescribirse para indicar al agente de forma manual el comportamiento y acciones a realizar en cada paso del entrenamiento.
    - Inference Only. Indica que el agente funcionará con el modelo asignado en el campo "Model". De no haber establecido ningún modelo, no podrá seleccionarse este tipo de comportamiento.

El resto de los parámetros son de menor importancia para el presente proyecto.

Para determinar el momento en el que el agente tiene que decidir durante el entrenamiento puede realizarse de dos maneras. Una es utilizando la función "RequestDecision ()" de la clase "Agent" de la cual extiende el agente. Llamando a esta función se comienza el ciclo del entrenamiento por parte del agente, es decir, partiendo de que el entorno se haya reiniciado.

La otra forma es añadiendo al agente el componente "Decision Requester" como aparece en la siguiente Figura 9:

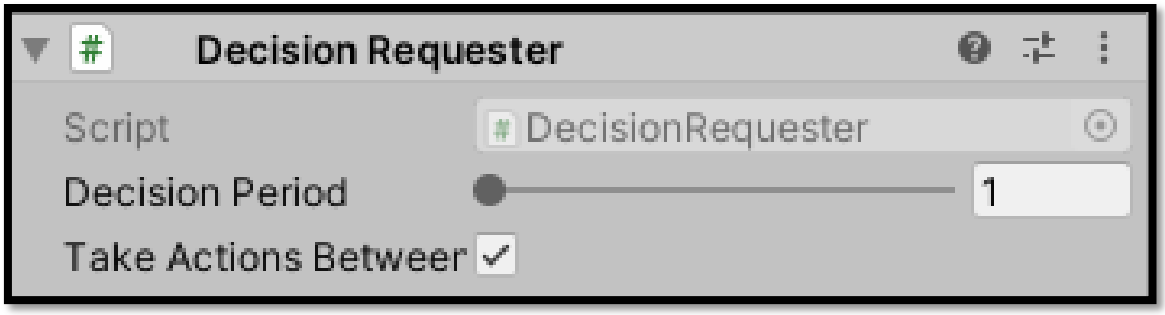


FIGURA 9. COMPONENTE "DECISION REQUESTER" DEL AGENTE.

Es aquí donde se indica cada cuantos pasos del entrenamiento el agente debe tomar decisiones y si debe realizar una acción después de tomar cada decisión.

Diseño del entorno de entrenamiento

Como se ha explicado en líneas anteriores del apartado, uno de los tipos de entrenamiento disponibles en la librería es el entrenamiento por Curriculum. Este tipo es especialmente importante para el proyecto, puesto que varios de los experimentos se han realizado usando este tipo de entrenamiento [9].

Por ello, se van a explicar cada uno de los parámetros que necesitan configurarse del Curriculum para que este pueda utilizarse.

En la siguiente Figura 10 puede verse un Curriculum de uno de los ejemplos de la librería.

```
curriculum:
  BigWallJump:
    measure: progress
    thresholds: [0.1, 0.3, 0.5]
    min_lesson_length: 100
    signal_smoothing: true
    parameters:
      big_wall_min_height: [0.0, 4.0, 6.0, 8.0]
      big_wall_max_height: [4.0, 7.0, 8.0, 8.0]
```

FIGURA 10. CURRICULUM DEL EJEMPLO "WALLJUMP" DE LA LIBRERÍA.

Una vez establecidos los hiper parámetros de la red neuronal necesarios para el entrenamiento de esta, se puede establecer un entrenamiento por Curriculum indicándolo en el archivo. Se

establece el “Behavior Name” o nombre de comportamiento (“BigWallJump” en el ejemplo de la Figura 10) de la política sobre la que se va a aplicar dicho entrenamiento, y se pasa a configurar cada uno de los parámetros.

- Measure. Establece el tipo de medida que se va a usar para avanzar en el Curriculum. Esta medida puede ser de tipo “progress” o progreso, o de tipo “reward” o recompensa.
- Tresholds. Son los límites establecidos sobre los que el Curriculum va avanzando. Si el tipo de medida elegido es de tipo progreso, los límites indican cada cuanto porcentaje de completitud del entrenamiento se debe avanzar. En el caso de ser de tipo recompensas, indica la recompensa que se debe alcanzar para avanzar.
- Min_lesson_length. Determina el número mínimo de iteraciones de entrenamiento antes de avanzar. En caso de que el agente consiga uno de los límites de forma muy rápida, primero se debe comprobar que haya cumplido el mínimo de iteraciones.
- Signal_smoothing. Permite aplicar un peso o normalización sobre el actual límite de la medida utilizando los valores anteriores del entrenamiento.
- Parameters. Corresponde a cada uno de los parámetros que se van a ir cambiando a lo largo del entrenamiento. En el caso del ejemplo de la Figura 10, se cambia la altura del muro que debe saltar el agente.

Tensorboard

En adición a la librería TensorFlow, esta posee una herramienta llamada TensorBoard. Con ella se permite el visionado de gráficos y estadísticas de los datos de los entrenamiento realizados con la librería ML-Agents y TensorFlow [18].

Estas estadísticas se dividen en estadísticas del entorno y estadísticas de la política. Los datos que proporcionan son especialmente interesantes en los casos de entrenamientos con Curriculum. Las gráficas y datos más importantes que pueden obtenerse son los siguientes:

- Cumulative Reward (Environment/Cumulative Reward). Representa la recompensa media a lo largo del entrenamiento. En un entrenamiento exitoso la gráfica debería mostrar una recompensa ascendente con respecto al avance del entrenamiento, aunque es dependiente de la complejidad del problema.
- Extrinsic Reward (Policy/Extrinsic Reward). Representa la recompensa acumulada conseguida por el agente a lo largo de cada iteración del entrenamiento.

- Entropy (Policy/Entropy). Representa la aleatoriedad de la toma de decisiones que realiza el agente. En un entrenamiento exitoso este valor debería ir disminuyendo según avance el entrenamiento.
- Policy Loss (Losses/Value Loss). Representa la magnitud de la política en materia de cambios. Es decir, la cantidad de veces que la política cambia durante el entrenamiento. En el caso de un entrenamiento exitoso, este valor debería ir disminuyendo conforme el entrenamiento avance, aunque el valor puede ir oscilando.
- Value Loss (Losses/Value Loss). Representa la capacidad del modelo de predecir el valor de la recompensa futura de cada iteración del entrenamiento. En un entrenamiento por Curriculum exitoso, este valor debería ir disminuyendo conforme el entrenamiento avance, aunque es posible que el valor oscile a lo largo de este.

Pandemic

"Pandemic" es un juego de mesa cooperativo de uno a cuatro jugadores donde tendrán que trabajar juntos para controlar, buscar y erradicar a un total de cuatro virus que brotan en distintas localizaciones del planeta. Cada uno de los jugadores desempeña un rol de personaje en el equipo a lo largo de la partida, donde cada rol posee habilidades especiales que ayudarán a en la misión de erradicar los virus y así conseguir la victoria [19].

Como se ha explicado el objetivo del juego es erradicar los virus que van surgiendo en cada turno en las regiones del planeta. Para ello, primero se deberá cooperar y encontrar una cura para cada uno de los virus. Debido a la complejidad del juego, se han utilizado unas reglas simplificadas para la adaptación. A continuación, se detallan las reglas usadas en la adaptación.

- Elementos para el juego:
    - Mapa del tablero. Se comprende de un total de 48 nodos de ciudades conectados de una forma específica.
    - Jugadores. Ficha que representa un jugador.
    - Mazo de Infección. Mazo que contiene las cartas que provocan el virus. Cada carta posee un nombre de ciudad donde colocar el virus.
    - Mazo del jugador. Mazo que contiene cartas neutrales (para simplificar el juego) que no tienen ningún efecto y cartas de epidemia (se explicará su función más adelante).

- Ratio de infección. Indica el número de cartas a robar al final del turno del jugador.
- Indicador de brotes. Contabiliza el número de brotes de la partida.

- Reglas de la partida:
  - Inicio del juego. Se colocará el jugador en el nodo de Atlanta y se generarán los virus iniciales. Para ello se roban un total de nueve cartas del mazo de infección cogiéndolas de tres en tres. En cada carta del grupo de tres se colocarán tres virus para el primer grupo, dos para el segundo y uno para el tercero. También se inicializan los indicadores de infección y brotes.
  - Turno del jugador. Después de reiniciar los elementos iniciales de la partida, comienza el turno del jugador. Cada turno, se tienen un total de cuatro acciones permitidas. Las acciones que se pueden realizar son moverse a un nodo o curar en el nodo actual, consumiendo un punto de acción cada una.
  - Final del turno del jugador. Al finalizar el turno, el jugador debe robar dos cartas del mazo de jugador. Además, se robarán tantas cartas de la cima del mazo de infección como se indique en el ratio de infección y se colocará un virus en la ciudad de la carta (si posee menos de tres virus, en caso contrario ocurre un Brote).
  - Epidemia. En el caso de robar una carta de epidemia del mazo del jugador, se incrementará una posición el ratio de infección y se robará una carta del final del mazo de infección. También se colocarán tres virus en la ciudad indicada (en caso de no poder colocar los tres virus, se colocarán hasta llegar a los tres y habrá un Brote). Al final de la epidemia se barajearán las cartas descartadas del mazo de infección y se colocarán en la cima del mazo de infección.
  - Brote. Cuando ocurre un brote en la partida, se incrementará el indicador de brote en uno. Además, se colocará un virus en las ciudades conectadas a la ciudad donde se ha originado el Brote (en caso de tener menos de tres virus, en caso contrario ocurre un Brote. Produciéndose en cadena).
  - Final del Juego. Puesto que las reglas están simplificadas, es imposible que se dé la condición para que el jugador gane la partida. Sin embargo, si el indicador de brotes llega al final, no quedan más virus que colocar o no quedan cartas de jugador que robar, la partida se da por perdida. No obstante, con los experimentos se ha buscado alargar la partida en favor del jugador, no completar una sesión de juego.

Por último, para concluir con el apartado del juego, se muestran a continuación los objetos y funciones que se han programado en el motor Unity.

En las siguientes Figura 11 y Figura 12, pueden verse los objetos de juego en Unity que se han creado en representación de los elementos usados para la partida.

FIGURA 11. OBJETOS DEL JUEGO USADOS EN UNITY.

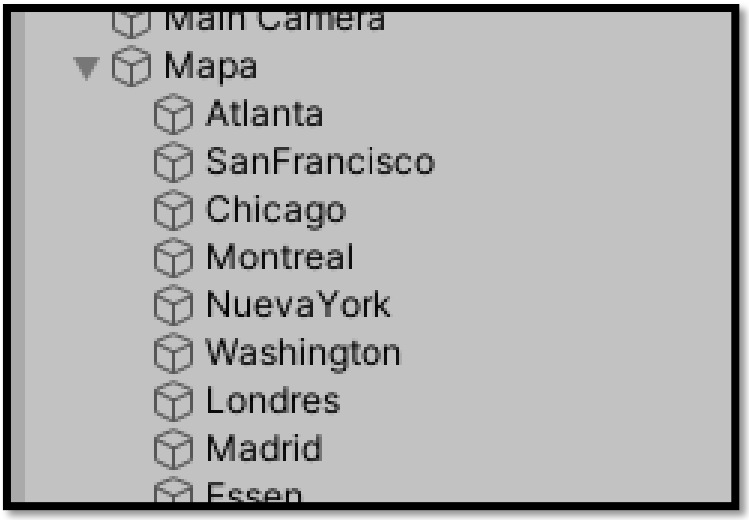


FIGURA 12. OBJETOS HIJOS EN REPRESENTACIÓN DE CADA NODO DEL MAPA.

También, la inclusión de una imagen de fondo para tener una referencia visual del mapa.

FIGURA 13. IMAGEN DEL TABLERO DONDE SE HAN COLOCADO EL RESTO DE LOS OBJETOS.

Como se puede comprobar se ha creado un mapa (puede verse en la Figura 13) en representación a todos los nodos que conforman el mapa original, un cilindro colocado en Atlanta en representación de la ficha de jugador y un fondo que representa el tablero.

Además, cada uno de estos objetos tiene como componente adjunto un script que representa sus clase con las funciones y datos necesarios para hacer funcionar al juego. Las clases y funciones que se han programado son las siguientes:

- Nodo. Representa a una ciudad individual. Posee como atributo un nombre de ciudad y el número de virus que hay en ella.
- Mapa. Representa el tablero y almacena todos los nodos. Posee como atributo una lista de los nodos y un diccionarios con las conexiones que formadas entre todos ellos. Además de algunas funciones para el manejo de estas listas.
- Carta. Representa una carta de ambos mazos. En el caso de ser de infección contiene el nombre de la ciudad, en el caso del jugador solo contiene si es de epidemia o no.
- BarajaI. Representa el mazo de cartas de infección. Contiene una lista de cartas con cada una de las ciudades que pueden infectarse y un color. También posee métodos para barajar el mazo o robar un carta de la cima o final del mazo.
- BarajaJ. Representa el mazo de cartas del Jugador. Contiene cartas de relleno sin valor para el jugador excepto las de epidemia que ocasionan este evento con su mismo nombre. También posee métodos para barajar el mazo o robar un carta de la cima o final del mazo.
- Jugador. Representa al jugador de la partida, es decir, el agente que va a entrenar y jugar la partida. Al ser la clase del agente posee como atributos toda la información relevante de la partida (como el nodo en el que se encuentra, las conexiones del nodo, etc.) además de tener las funciones necesarias para entrenar que se han detallado en el apartado de “Experimentos”.
- Juego. Clase que representa la lógica de reglas. Esta clase engloba todos los elementos del juego simulando al tablero. Además, tiene todas las funciones programadas para que el juego funcione según las reglas adaptadas explicadas anteriormente.

Experimentos

Movimiento del agente por camino más corto

Una vez se han adquirido los conocimientos básicos sobre el funcionamiento de Unity y ML-Agents y se tiene una base programada de los objetos del juego, se ha pasado desarrollar los primeros pasos para el agente, véase, el movimiento.

Para hacer más sencillo el problema, se ha comenzado por un movimiento a vecinos inmediatos, es decir, que el agente sea capaz de moverse a un nodo adyacente.

Como se ha comentado en la parte anterior, el flujo de aprendizaje de la librería sería reiniciar el entorno, recoger información del estado del entorno, que el agente tome una decisión de acción, recibir una recompensa y volver a reiniciar el entorno, todo esto en bucle el número de iteraciones establecidas.

El objeto y clase "Jugador" es entonces el agente encargado de realizar el entrenamiento. Para poder realizarlo se han programado las siguientes funciones:

- "OnEpisodeBegin ()". En ella se reinicia el entorno de entrenamiento. Como se trata del movimiento, al reiniciar el entorno se le dado al agente un nodo aleatorio como nodo a partir, y colocado el objeto de Unity en él. También se inicializa la lista de nodos vecinos y se coge uno de ellos como nodo destino.
- "CollectObservations (VectorSensor sensor)". En ella se recoge la información del estado del entorno. Al tratarse del movimiento, se recoge información del nodo actual del agente y de cuál es el nodo de destino.
- "CollectDiscreteActionMasks (DiscreteActionMasker actionMasker)". En ella se enmascaran las acciones que puede escoger el agente. Puesto que en el movimiento solo puede moverse a nodos adyacentes, se enmascaran todas las acciones permitiendo así solo a los nodos vecinos.
- "OnActionReceived (float [] vectorAction)". Función que ejecuta la acción escogida por el agente. En este caso, se recoge el nodo vecino escogido por el agente y se le desplaza a él. Además, se esta función también ejerce como la función de recompensa. Es decir, la función donde se trata de recompensar o castigar al agente según la decisión tomada.

Al tratarse esta última de prácticamente la función más importante para el entrenamiento, se va a analizar en detenimiento.

```
3 referencias
public override void OnActionReceived(float[] vectorAction)
{
    nodoActual = mapa.GetNodos()[Mathf.FloorToInt(vectorAction[0])];

    if (nodoActual == nodoObjetivo)
    {
        AddReward(1.0f);
        EndEpisode();
    }
    if (MaxStep == 0)
    {
        AddReward(-0.1f);
        EndEpisode();
    }
}
```

FIGURA 14. FUNCIÓN DE RECOMPENSA DE MOVIMIENTO POR CAMINO MÁS CORTO SIMPLE.

Como se puede comprobar primero se establece como nodo actual el escogido como destino por el agente del vector de acciones para que se desplace. Seguidamente se le otorgan las recompensas específicas dependiendo del resultado. Si se escoge como nodo de destino al nodo objetivo, se le añade a la recompensa un punto. De no se ser así escogerá otro nodo. Para terminar, si se le agota el máximo de acciones permitidas para un turno del juego (cuatro acciones), se le aplica un pequeño castigo restándole una décima de punto.

Con esta lógica, el agente buscará obtener siempre la mejor recompensa, siendo a medida que avanzan las iteraciones, más alta su tasa de aciertos en llegar al destino.

Antes de pasar al entrenamiento se han de configurar los parámetros del componente “Behavior Parameters” que se le añade al agente para controlar el comportamiento de la política. Para este experimento se ha configurado según muestra la siguiente Figura 15:

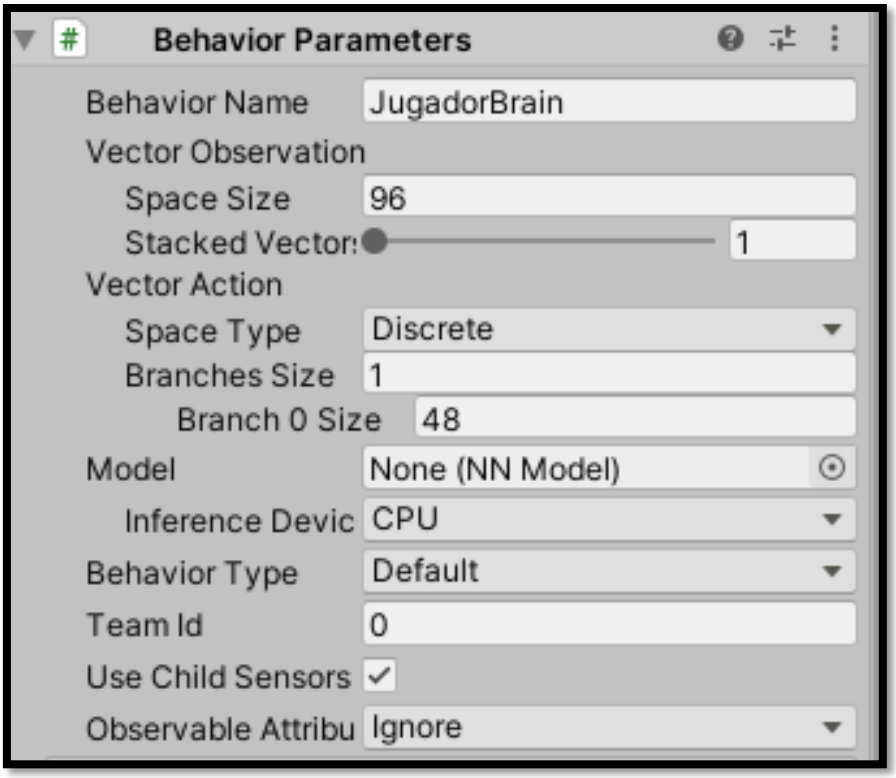


FIGURA 15. COMPONENTE "BEHAVIOR PARAMETERS" PARA EL MOVIMIENTO POR CAMINO MÁS CORTO SIMPLE.

- Vector Observation
    - Space size. Se ha establecido un espacio de tamaño 96. Esto corresponde a dos vectores de un tamaño de 48 cada uno recogidos con la función "CollectObservations (VectorSensor sensor)".
    - Stacked Vector. Establecido a uno. Es decir, solo se usa la información de la iteración del entrenamiento actual.
- Vector Action
    - Space type. Se ha seleccionado un espacio de tipo discreto.
    - Branches size. Tamaño de 48 nodos que corresponden a los nodos a enmascarar.

El modelo de red no se ha aplicado, puesto que es necesario entrenarlo primero. Y en cuanto al tipo de comportamiento, se ha seleccionado por defecto y que sea Unity el que lo cambie automáticamente.

También, el momento de pedir decisión al agente se ha realizado añadiendo el componente "Decision Requester" explicado y especificando que tome decisiones en cada intervalo. Puede verse el componente en la Figura 16.

FIGURA 16. COMPONENTE "DECISION REQUESTER" PARA EL MOVIMIENTO POR CAMINO MÁS CORTO SIMPLE.

Programadas las funciones y establecidos los parámetros de la política del agente, se ha procedido a realizar el primer entrenamiento del agente. Para visualizar los resultados del entrenamiento, a continuación, se muestran una serie de gráficas proporcionadas por la herramienta TensorBoard para ver los valores obtenidos.

- Cumulative Reward. Se puede comprobar la recompensa obtenida por el agente a lo largo del entrenamiento. Se consigue una recompensa alta que va ascendiendo desde un principio a lo largo de las iteraciones. Esta se mantiene estable con un valor de 0.7903 sobre un máximo de 1 después de 170000 pasos.

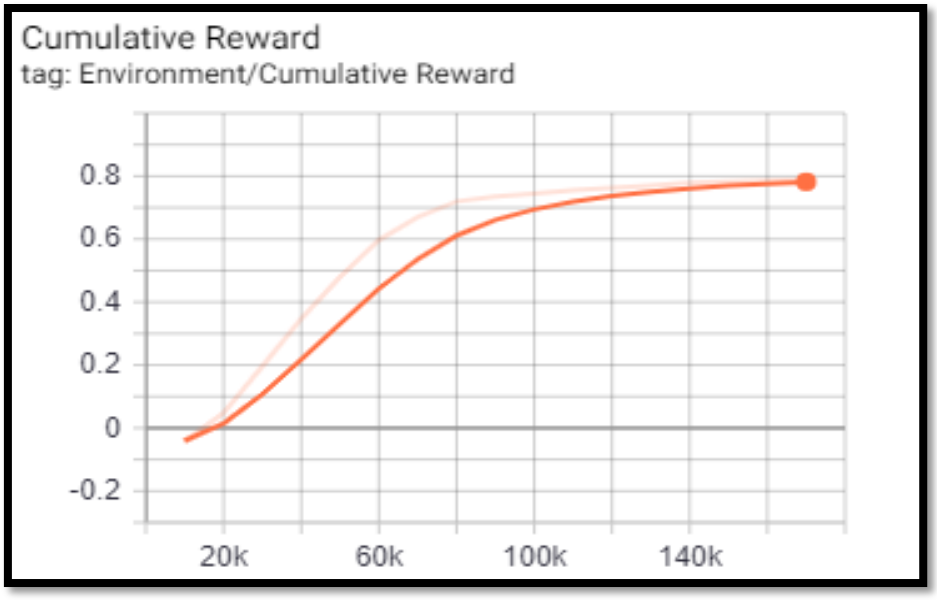


FIGURA 17. GRÁFICA "CUMULATIVE REWARD" PARA EL MOVIMIENTO MÁS CORTO.

- Entropy. Como puede observarse, la aleatoriedad con la que el agente toma decisiones va disminuyendo conforme el entrenamiento avanza hasta alcanzar un mínimo de 0.2521 al final de todos los pasos.

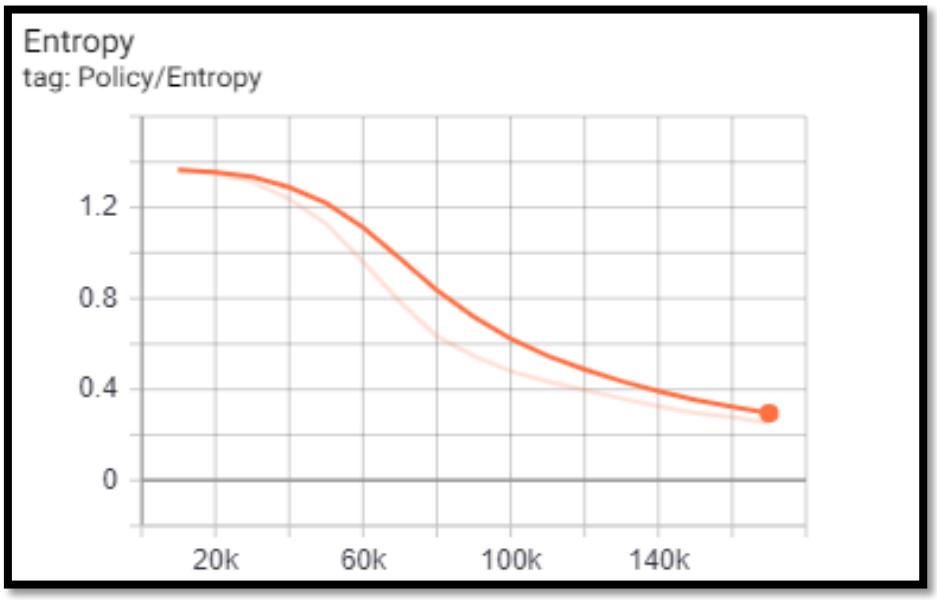


FIGURA 18. GRÁFICA "ENTROPY" PARA EL MOVIMIENTO MÁS CORTO.

- Policy Loss. Se puede apreciar que la política comienza con un valor sobre 0.1409 y que va en ascenso al comienzo del entrenamiento. Conforme se alcanza el ecuador los valores empiezan a disminuir hasta estabilizarse en una política sin muchos cambios.

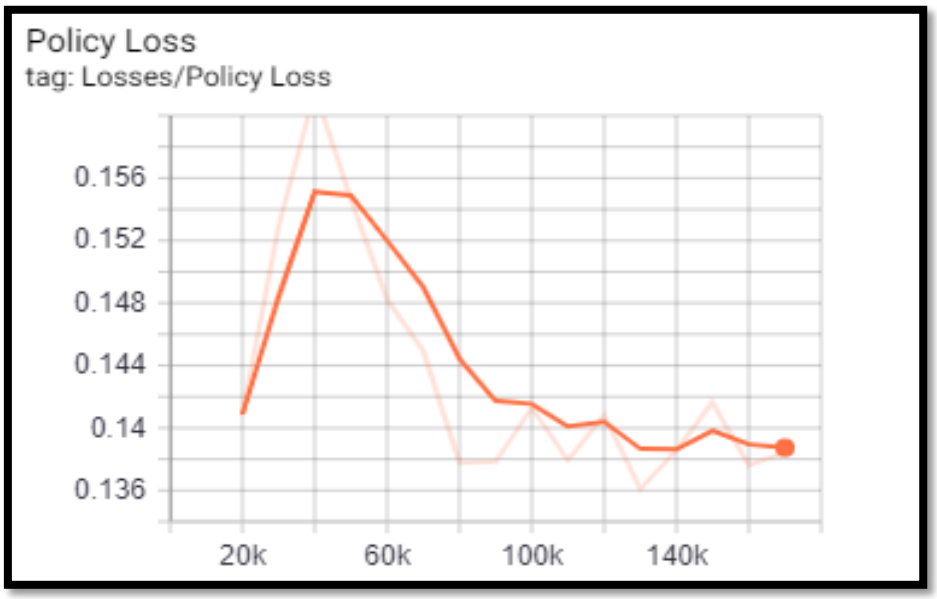


FIGURA 19. GRÁFICA "POLICY LOSS" PARA EL MOVIMIENTO MÁS CORTO.

- Value Loss. En esta gráfica, se comienza con un valor de 0.304 que indica una predicción escasa del agente de la recompensa futura. Según el entrenamiento avanza, este valor decrece hasta establecerse un valor final de 0.06947.

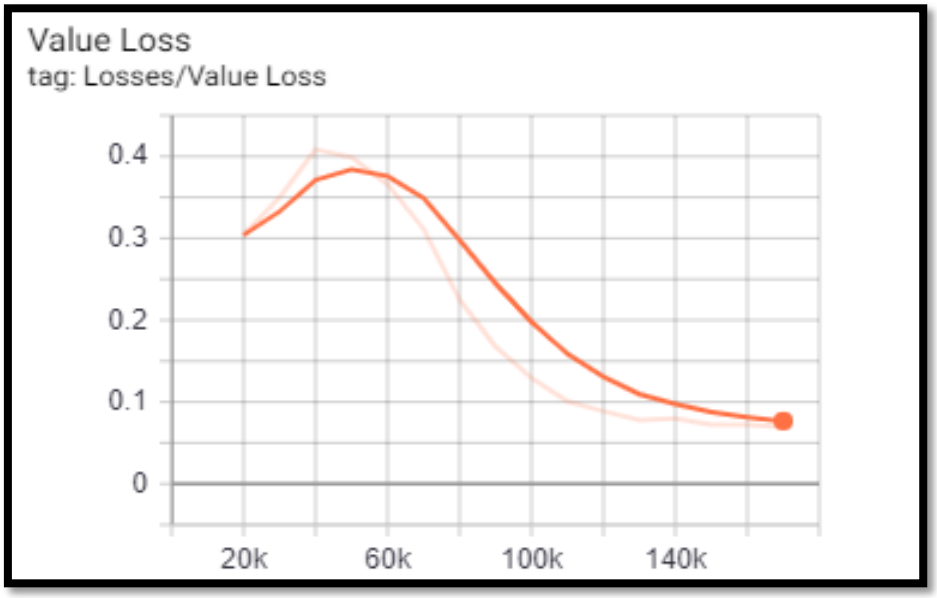


FIGURA 20. GRÁFICA "VALUE LOSS" PARA EL MOVIMIENTO MÁS CORTO.

Completado el movimiento más simplificado, se ha pasado al movimiento normal con rutas más altas donde el agente pueda moverse durante el turno. Esto es, rutas de una distancia de cuatro saltos o más. Usando las mismas funciones que se han programado con el movimiento anterior la tasa de acierto del entrenamiento oscila entre valores negativos y positivos, lo que supone no completar el entrenamiento satisfactoriamente ni llegar prácticamente en ninguna ocasión a completar saltos de altas distancias.

Para encontrar una solución a este problema, se ha usado uno de los tipos de entrenamiento de la librería para entornos específicos. En este caso, se ha implementado el entrenamiento por Curriculum.

Entrenamiento por Curriculum

Para usar este tipo de entrenamiento, se ha de especificar en el archivo Curriculum con los parámetros de entrenamiento qué partes del entorno van a ser las cambiantes. A continuación, puede verse en la Figura 21 las reglas del Curriculum.

```
measure: reward
thresholds: [0.9, 0.93, 0.95, 0.97, 1]
min_lesson_length: 100
signal_smoothing: true
parameters:
    dist_ruta: [1, 2, 3, 4, 5, 6]
```

FIGURA 21. CURRICULUM PARA EL MOVIMIENTO POR CAMINO MÁS CORTO NORMAL.

Se ha establecido que el parámetro del entorno cambie con una medida por recompensa, es decir, cuando el agente vaya alcanzando las recompensas medias establecidas en el campo "tresholds". Cada recompensa alcanzada del campo "tresholds", aumentará la distancia máxima de rutas que pueden aparecer en el entorno durante el entrenamiento. Debido a la formación del mapa del juego, las rutas de una distancia de seis saltos son suficientes para recorrerlo al completo. Con el Curriculum especificado, se ha vuelto a probar el entrenamiento del agente. Esta vez se ha conseguido completar el entrenamiento y con tasas de aciertos altas con un valor de 0.9455 mayores a las del entrenamiento simple como puede verse en la siguiente gráfica:

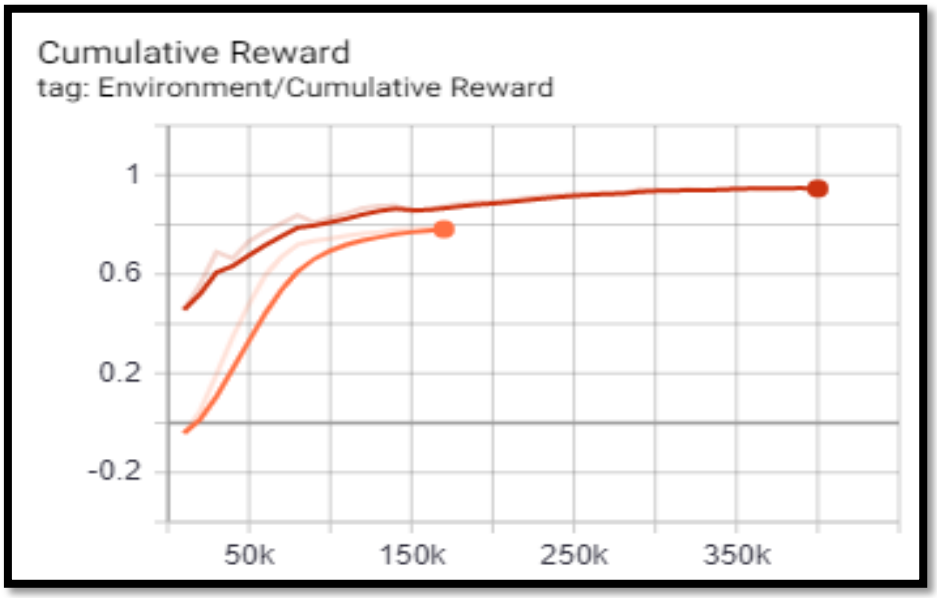


FIGURA 22. GRÁFICA "CUMULATIVE REWARD" DEL MOVIMIENTO POR CAMINO MÁS CORTO NORMAL (ROJO) FRENTE AL MOVIMIENTO POR CAMINO MÁS CORTO SIMPLE (NARANJA).

Habiendo completado el movimiento por camino más corto normal del agente, y, por consiguiente, el objetivo de este experimento, se han investigado otras opciones de entrenamiento que ayuden al agente a encontrar antes la recompensa óptima.

Optimización del programa

Una de las formas que se han investigado es la aplicación de recompensas negativas en cada iteración del entrenamiento. A continuación, puede verse en la Figura 23 como se ha programado una nueva función de recompensa con dicha técnica:

```
3 referencias
public override void OnActionReceived(float[] vectorAction)
{
    AddReward(-1f / MaxStep);

    nodoActual = mapa.GetNodos()[Mathf.FloorToInt(vectorAction[0])];

    if (nodoActual == nodoObjetivo)
    {
        AddReward(1f);
        EndEpisode();
    }
}
```

FIGURA 23. FUNCIÓN DE RECOMPENSA DEL MOVIMIENTO POR CAMINO MÁS CORTO CON RECOMPENSAS NEGATIVAS.

En este caso, se ha aplicado en cada iteración una pequeña recompensa negativa o castigo al agente. Esto ha supuesto una aceleración en el proceso de aprendizaje del agente, puesto que, a más iteraciones en alcanzar el nodo, peor será la recompensa total. Además, dado que se aplica un castigo en cada iteración, se ha eliminado la restricción de realizar el movimiento en un número máximo de acciones por turno como estaba programado en la función original (Figura 14).

Movimiento del agente por curación máxima automática

Una vez conseguido el movimiento del agente por el camino más corto, se ha pasado a programar que el agente priorice la curación al moverse por los nodos. Es decir, que el agente se mueva a los nodos infectados y los cure automáticamente una vez está allí situado. Terminando por curar todos los nodos existentes de la forma más óptima.

Para realizar este cometido, se ha utilizado la lógica y funciones programadas en el apartado anterior, realizando solo algunos cambios en las funciones del entrenamiento y añadiendo alguna función para el manejo de los datos.

En cuanto a las funciones del manejo de los datos, se han programado dos funciones en adición a la clase original.

- Una función “GeneraVirus ()” para generar virus por el mapa en cada iteración con el cometido de no depender de las reglas del juego y poder realizar el entrenamiento.
- Una función “SinVirus ()” que, simplemente, compruebe la existencia de virus en el mapa. Necesaria para verificar el final de una iteración donde se hayan curado todos los virus generados.

Por otro lado, los cambios que se han realizado a las funciones de entrenamiento son los siguientes:

- “OnEpisodeBegin ()”. Al tratarse ahora de moverse por una curación máxima automática, al reiniciar el entorno se le ha vuelto a dar al agente un nodo aleatorio como nodo a partir, y colocado el objeto de Unity en él. Pero en este caso, se ha eliminado el generar un nodo de destino. En su lugar, se generan de uno a tres virus en varias localizaciones del mapa.
- “CollectObservations (VectorSensor sensor)”. Al tratarse de moverse para curar, se recoge información de los nodos disponibles para viajar y en vez del nodo de destino, información de los nodos que tienen virus.
- “CollectDiscreteActionMasks (DiscreteActionMasker actionMasker)”. Puesto que se sigue tratando en general de movimiento, se ha dejado el enmascarado original de moverse a nodos adyacentes, donde se enmascaran todas las acciones permitiendo así solo movimiento a los nodos vecinos.
- “OnActionReceived (float [] vectorAction)”. Al igual que antes, se recoge el nodo vecino escogido por el agente y se le desplaza a él. Pero puesto que el cometido del entrenamiento ha cambiado, la función de recompensa también. Se va a detallar a continuación la nueva función de recompensa programada.

```
3 referencias
public override void OnActionReceived(float[] vectorAction)
{
    AddReward(-1f / MaxStep);

    nodoActual = mapa.GetNodos()[Mathf.FloorToInt(vectorAction[0])];

    if (nodoActual.nVirus == 1)
    {
        nodoActual.nVirus -= 1;
        AddReward(0.25f);
    }
    if (nodoActual.nVirus == 2)
    {
        nodoActual.nVirus -= 2;
        AddReward(0.4f);
    }
    if (nodoActual.nVirus == 3)
    {
        nodoActual.nVirus -= 3;
        AddReward(0.55f);
    }
}
```

FIGURA 24. FUNCIÓN DE RECOMPENSA DEL MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

Se puede comprobar que se mantiene la forma de optimización del entrenamiento del experimento anterior aplicando un castigo en cada iteración. Una vez se ha escogido como antes el nodo al que quiere desplazarse el agente, se establece toda la nueva lógica de recompensas. Dependiendo del número de virus que posea el nodo, se le otorga al agente mayor o menor recompensa.

Después de haber realizado varios entrenamientos, se ha encontrado que los mejores valores como recompensa son los expuestos en la Figura 24 anterior. Dando mayor recompensa si se cura un nodo con tres virus, y descendiendo cuanto menor número de virus posea el nodo.
En el caso de no tener ningún virus, el agente se sigue desplazando por el mapa en busca de nodos con virus hasta haber terminado curando todos los nodos. Es entonces cuando la iteración del entrenamiento finaliza.

Se puede apreciar también la diferencia con la función del experimento anterior donde se controlaba que se alcanzase el objetivo en las cuatro acciones disponibles para el agente en cada

turno de la partida. En el caso de curar, es imposible realizar curaciones de todos los nodos que aparecen en un total de cuatro acciones máximas por turno, por lo que se ha eliminado esta restricción del entrenamiento. En su defecto, se verá en el siguiente apartado como, una vez entrenada la red, si se aplican el máximo de acciones, el agente realizará las cuatro primeras acciones del total para realizar la curación máxima.

Antes de pasar al entrenamiento, como en el experimento anterior, se han configurado los parámetros del componente "Behavior Parameters" para controlar la política. Se han fijado como en la siguiente Figura 25:

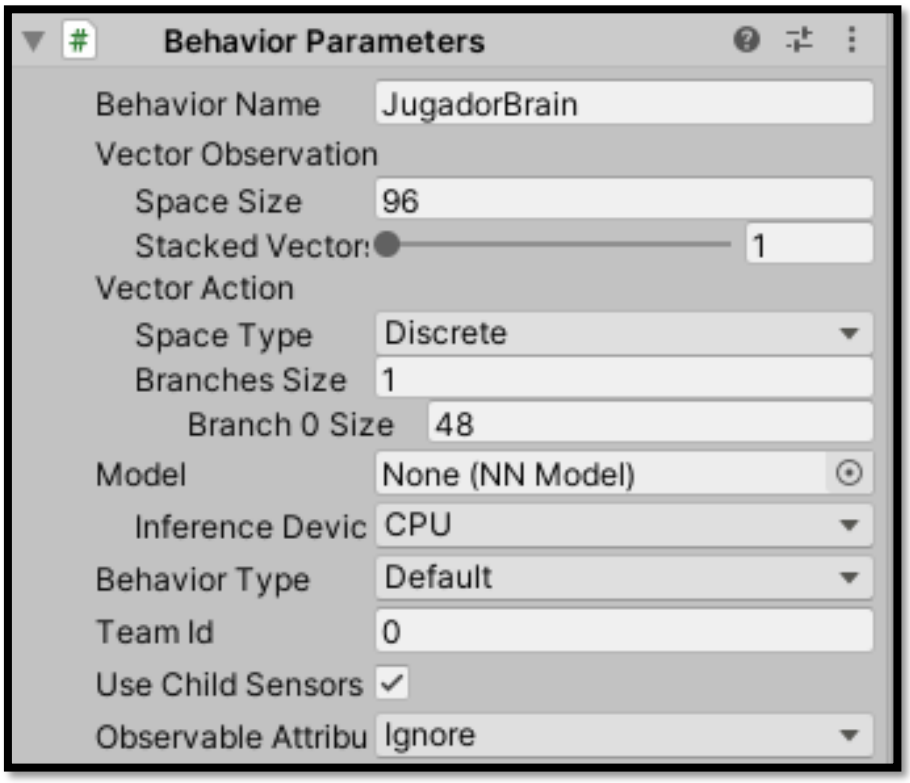


FIGURA 25. COMPONENTE "BEHAVIOR PARAMETERS" PARA EL MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

Se puede comprobar que coincide con los del experimento anterior:

- Vector Observation
    - Space size. Se ha establecido un espacio de tamaño 96. Esto corresponde a dos vectores de un tamaño de 48 similar al anterior, pero con información diferente recogida con la función "CollectObservations (VectorSensor sensor)".

    - Stacked Vector. Al igual que antes, establecido a uno. Es decir, solo se usa la información de la iteración del entrenamiento actual.
- Vector Action
    - Space type. Se ha seleccionado un espacio de tipo discreto.
    - Branches size. Tamaño de 48 nodos que corresponden al total de todos los nodos a enmascarar.

También, se ha utilizado el componente "Decision Requester" con un intervalo de decisión de uno. Similar al caso anterior. La diferencia ha sido la eliminación de la restricción de cuatro acciones como se ha explicado anteriormente. Para este experimento se ha usado un máximo de 30 pasos. De esta manera, junto con los ajustes de los valores de las recompensas, puede pasarse al entrenamiento.

FIGURA 26. MÁXIMO DE PASOS PARA EL MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

Una vez se tiene la nueva función y los parámetros establecidos, se ha utilizado para el entrenamiento los mismos métodos de aprendizaje que en el primer experimento. Es decir, al tratarse de una tarea aún más compleja que el movimiento por camino más corto normal, se ha usado el entrenamiento por Curriculum directamente como método de aprendizaje.

Usando un Curriculum similar al anterior, con medida de avance en el entrenamiento por alcanzar una recompensa, no se ha conseguido que el agente termine el entrenamiento. Quedándose estabilizada la recompensa aproximadamente con un valor de 0.9 a mitad de los parámetros del Curriculum con distancias de generación de virus medias y un número de virus medio. Que, aun siendo una recompensa alta, no se ha conseguido que avance y complete el Curriculum.

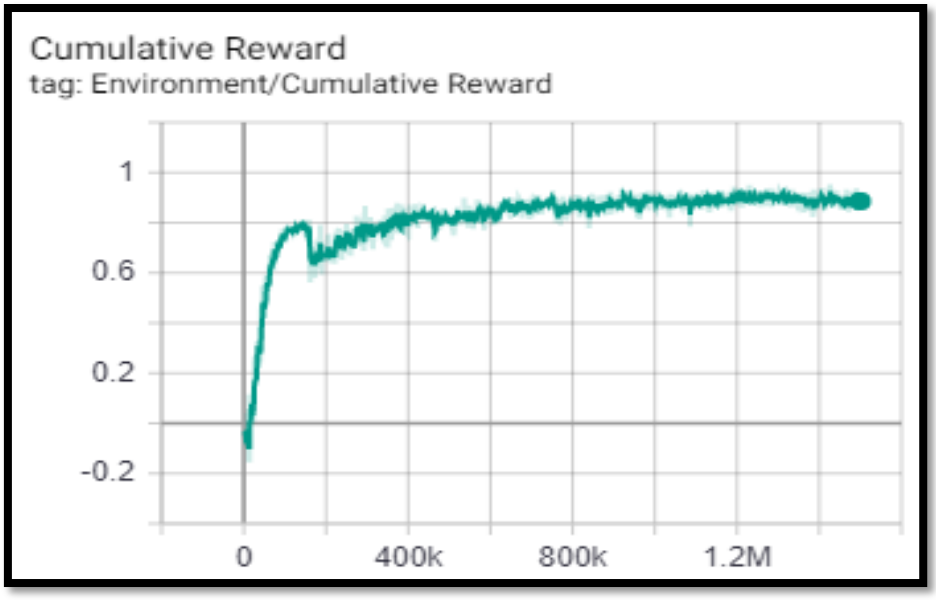


FIGURA 27. GRÁFICA "CUMULATIVE REWARD" CON CURRICULUM DEL PRIMER EXPERIMENTO.

Por lo que se ha optado por cambiar la medida de avance y expandir el número de valores de los parámetros para que la curva de aprendizaje sea más suave. Entonces, el Curriculum actualizado y que se ha especificado para este experimento es el de la siguiente Figura 28:

```
curriculum:
  JugadorBrain:
    measure: progress
    thresholds: [0.03, 0.06, 0.09, 0.125, 0.16, 0.2, 0.235, 0.27, 0.305, 0.345, 0.385,
                 0.43, 0.47, 0.51, 0.55, 0.595, 0.65, 0.7, 0.745, 0.79, 0.835, 0.89, 0.94]
    min_lesson_length: 1000
    signal_smoothing: false
    parameters:
      dist_ruta: [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
      virus_max: [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3 ,3 ,3 ,3, 4, 4, 4, 4, 4, 4]
```

FIGURA 28. CURRICULUM PARA MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

Se ha cambiado que la medida de avance para que cambien los parámetros del entorno sea por porcentaje del progreso del entrenamiento en vez de por recompensa media como en el primer experimento. Aproximadamente se avanza cada tres a cinco por ciento de los pasos del total del entrenamiento, dependiendo la fase dónde se encuentre el agente. Esto hace que al comienzo siendo la tarea más sencilla, se requieran menos iteraciones del total, y a medida que avanza y se complica, se da un poco más de margen de iteraciones.

Con esta actualización del Curriculum, se ha pasado a realizar de nuevo el entrenamiento. Esta vez, completando el entrenamiento comparado con el entrenamiento del Curriculum anterior.

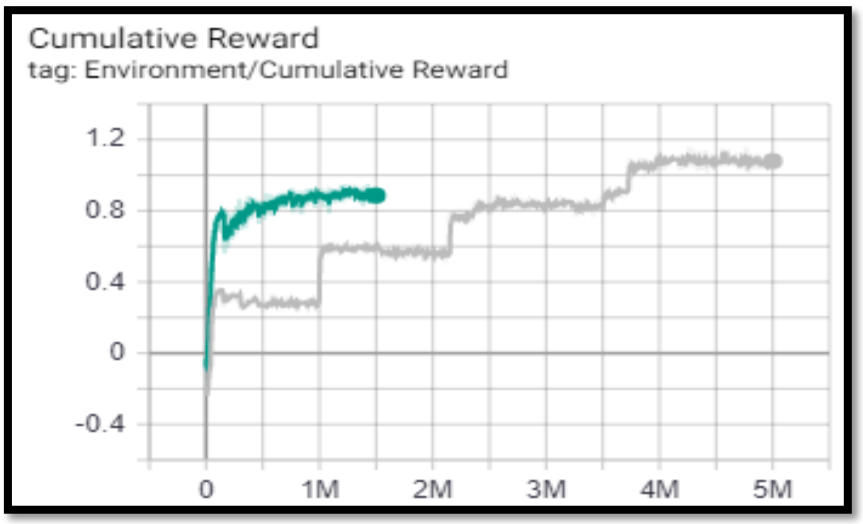


FIGURA 29. GRÁFICA "CUMULATIVE REWARD" COMPARANDO EL ENTRENAMIENTO. (EN VERDE EL PRIMER CURRICULUM Y EN GRIS EL ACTUALIZADO).

También, se han obtenido los siguientes resultados mostrados por la siguientes gráficas:

- Cumulative Reward. Se puede comprobar la recompensa obtenida por el agente a lo largo del entrenamiento. Cada escalón corresponde a una etapa del Curriculum en cuanto a los nodos con virus máximos que trata de curar. Al final del entrenamiento, para una distancia máxima de seis saltos y cuatro nodos con virus como máximo, se obtiene una recompensa de aproximadamente uno.

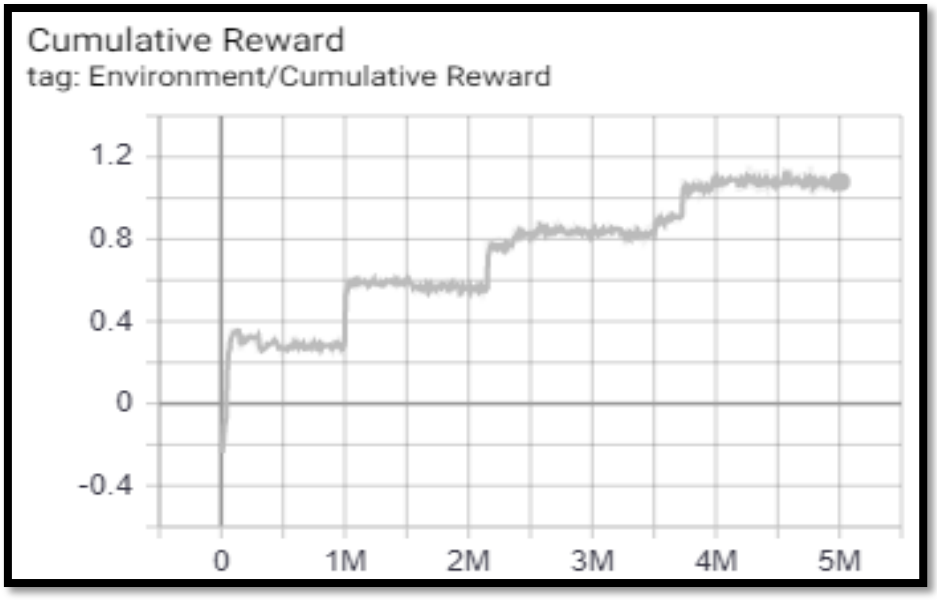


FIGURA 30. GRÁFICA "CUMULATIVE REWARD" PARA EL MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

- Entropy. Puede observarse que la aleatoriedad con la que el agente toma decisiones oscila en los primeros tramos del entrenamiento, pero en seguida va disminuyendo y manteniéndose estable hasta finalizar con un valor de 0.2032.

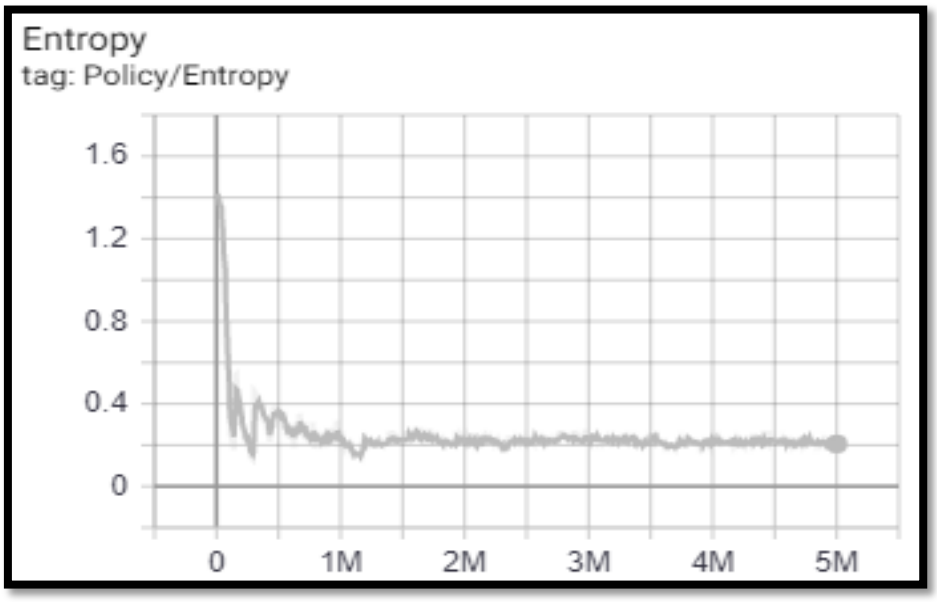


FIGURA 31. GRÁFICA "ENTROPY" PARA EL MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

- Policy Loss. En el caso de este experimento, puede verse como los valores van oscilando prácticamente todo el entrenamiento, lo que indica cambios en la política

continuamente. Sin embargo, a medida que avanza en las iteraciones el intervalo de los valores de oscilación disminuye, siendo el cambio de la política despreciable con un valor de 0.14.

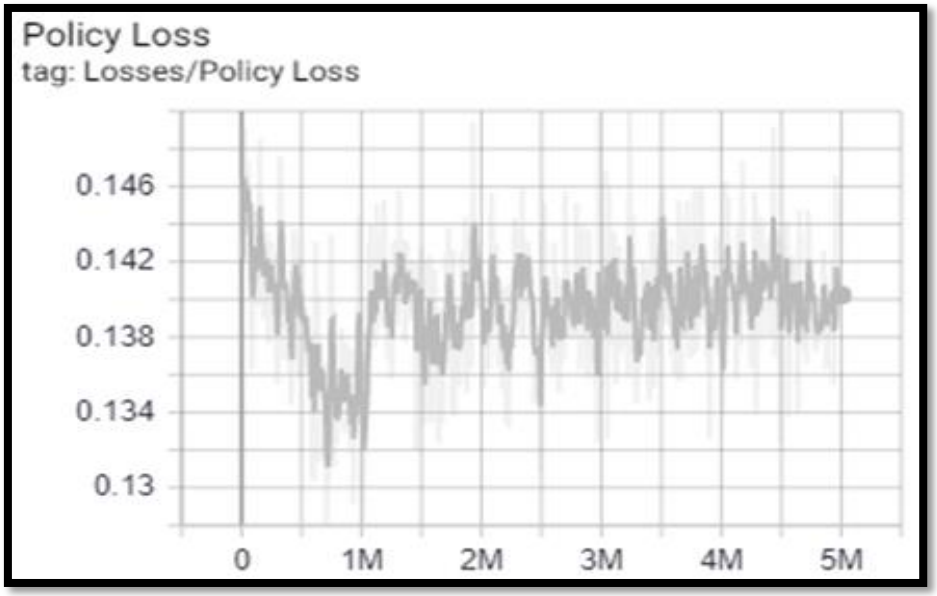


FIGURA 32. GRÁFICA "POLICY LOSS" PARA EL MOVIMIENTO MÁS CORTO.

- Value Loss. En esta gráfica, se comienza con un valor de 0.07368 que oscila hasta aproximadamente la mitad del entrenamiento. Lo que deriva en predicciones de la recompensa futura por parte del agente erróneas en ocasiones. Según el entrenamiento avanza, este valor decrece y se estabiliza hasta establecerse en un valor final de 0.03403.

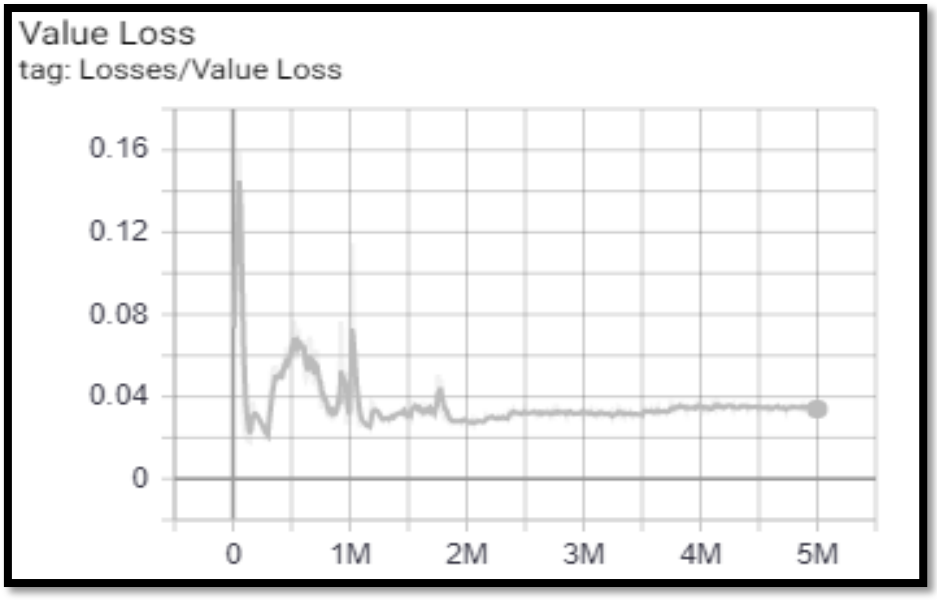


FIGURA 33. GRÁFICA "VALUE LOSS" PARA EL MOVIMIENTO MÁS CORTO.

Optimización del programa

Para minimizar el tiempo de entrenamiento de la red, se ha aplicado el método de entrenamiento multi agente, esto es, usando varios agentes entrenando y compartiendo los mismos datos bajo el mismo modelo de la red neuronal. Con esto se ha minimizado el tiempo y la rapidez de entrenamiento en general comparado con el entrenamiento de un solo agente. Pasando de un entrenamiento con un tiempo de 10m y 40s a un tiempo de 06m y 25s.

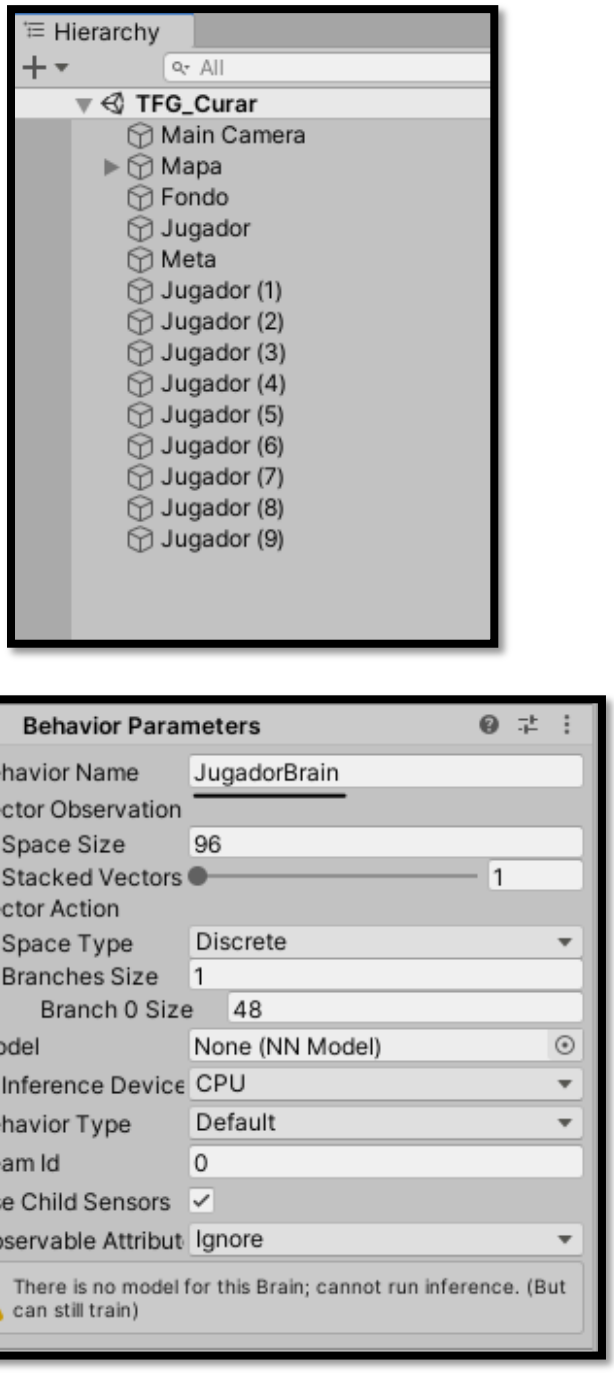


FIGURA 34. ENTRENAMIENTO MULTI AGENTE BAJO EL MISMO MODELO ("BEHAVIOR NAME").

Problemas y dificultades

Como se ha explicado en el experimento, uno de los problemas encontrados durante la realización de este, es la manera con la cual entrenar al agente. Usando un Curriculum algo más sencillo como el del primer experimento, el agente no tenía un entrenamiento suficiente ni con una escalabilidad de dificultad suave.

Esto se debe a que la red en cada entrenamiento se entrena desde cero, por lo que para moverse por curación máxima el agente primero debe aprender a moverse, como su propio nombre indica. Por ello se ha especificado un nuevo Curriculum donde se puede ver en el siguiente fragmento (Figura 35) la parte del movimiento.

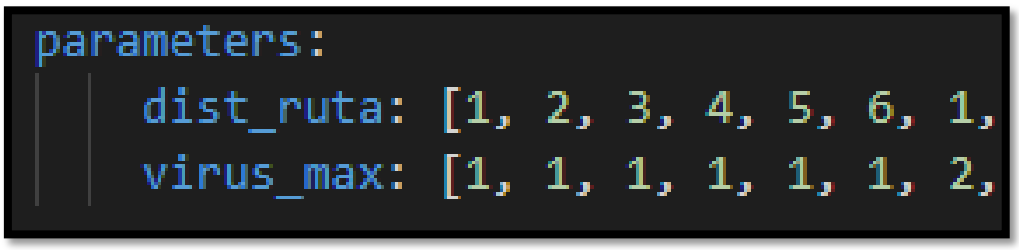


FIGURA 35. FRAGMENTO DEL CURRICULUM DE MOVIMIENTO POR CURACIÓN MÁXIMA AUTOMÁTICA.

Al ser solamente un virus el objetivo a curar, en estos primeros pasos el agente aprende a moverse de una manera lo más optima posible, parecida al primer experimento por el camino corto. Esto ocurre porque independientemente del número de virus del nodo, el agente recibe obligatoriamente una recompensa positiva al curar el nodo, por lo que la recompensa es mayor cuanto antes llegue.

Por ello, una vez aprende a moverse prácticamente por el camino más corto, se van aumentando el número de virus como lo especificado anteriormente. Para esta vez, moverse por una curación máxima del total de los virus.

Decisión del agente entre movimiento o curación

En este último experimento antes de pasar a las pruebas de comportamiento del agente entrenado en una partida del juego, se ha investigado como entrenar al agente en la elección

mezcla de los dos primeros experimentos. Es decir, se ha tratado de que el agente no solo se mueva de forma óptima y cure también, si no que en adición tome la decisión de cuál de las dos acciones es más conveniente en cada momento del entrenamiento y otorga más recompensa.

Como se ha comentado en el experimento previo y viendo los problemas surgidos durante este, es necesario que el agente aprenda a moverse antes de decidir si mover o curar, de lo contrario el entrenamiento nunca es completado. Para ello una de las formas es la especificación de un Curriculum como en el caso anterior, para que una vez aprendido el movimiento el agente ya pueda aprender a curar.

Pero dada la dificultad extra de la toma de decisión entre dos acciones, se ha optado por calcular de forma directa con un algoritmo de cálculo del camino más corto las distancias al nodo de destino elegido por el agente, contabilizando solo las acciones que el agente usa en el movimiento hasta el destino y no siendo necesario que este aprenda el camino más corto hasta él.

Así pues, se han planteado dos enfoques para el entrenamiento del agente que son los siguientes:

- Entrenamiento con un movimiento local. En este entrenamiento se calculan las distancias y a la hora de realizar un movimiento se hace igual que en los anteriores experimentos, moviéndose de nodo a nodo.
- Entrenamiento con un movimiento global. En este caso, igualmente se calculan las distancias, pero el movimiento se realiza directamente viajando al nodo de destino y descontando el número de saltos para llegar.

Con este objetivo y siguiendo la dinámica de los anteriores experimentos, se ha comenzado por el diseño de las funciones común a los dos enfoques con las que el agente va a funcionar. Para ello, se han actualizado y programado de la siguiente manera.

En cuanto a las funciones para la generación y manejo de los datos del entorno, se han programado tres funciones en adición a la clase original.

- Función “CalcularDistancias ()”. Se inicializan todas las distancias entre los 48 nodos del mapa ayudándose de la función auxiliar “CalcularDistanciasRec ()”.
- Función “CalcularDistanciasRec ()”. Función auxiliar que calcula de forma recursiva las distancias desde un nodo pasado al resto de nodos del mapa.
- Función “DistanciasNodo ()”. Devuelve las distancias de un nodo pasado al resto de nodos del mapa.

Por otro lado, los cambios que se han realizado a las funciones de entrenamiento se detallan a continuación. Solamente diferenciándose los dos enfoques en la recogida de información y enmascaramiento de las acciones.

- “OnEpisodeBegin ()”. Al tratarse este experimento de decidir entre las dos acciones de los experimento anteriores, al reiniciar el entorno se le ha vuelto a dar al agente un nodo aleatorio como nodo a partir, y colocado el objeto de Unity en él. También, se sigue manteniendo la generación de uno a tres virus en varias localizaciones del mapa.
- “CollectObservations (VectorSensor sensor)”.
    - Movimiento local. Siendo este experimento el más complicado de todos, la información que se le ha pasado al agente ha sido más extensa. Además de la información anterior como el nodo en el que se encuentra, los nodos adyacentes al actual y los nodos con virus, se ha añadido el número de virus que tiene cada uno y la información del nodo actual.
    - Movimiento global. Por el contrario, en este enfoque se ha recogido información del nodo actual donde se encuentra, los nodos con virus y los virus que tiene cada uno, y en este caso, las distancias del nodo actual al resto de nodos y el número de acciones que el quedan por realizar al agente. De esta manera, el agente puede saber que nodo queda en su alcance.
- “CollectDiscreteActionMasks (DiscreteActionMasker actionMasker)”.
    - Movimiento local. A la hora de enmascarar las acciones, es aquí donde el agente decidirá si moverse o curar. Para ello, la rama de acciones se ha aumentado en una acción más, que correspondería a curar. Quedando un total de 49 acciones. Es decir, curar o moverse a cualquiera de los nodos restantes disponibles. Siendo en este caso los adyacentes.
    - Movimiento global. Para el enfoque global, el espacio del vector es el mismo, de tamaño 49. Siendo una acción curar y el resto moverse a cualquiera de los

nodos restantes disponibles. Que, en este caso, son los alcanzables por las acciones restantes.

- "OnActionReceived (float [] vectorAction)". En esta ocasión, y al igual que en los dos experimentos anteriores, esta función se usa como función de recompensa a parte de ejecutar las acciones. Debido al experimento, se detalla a continuación los cambios realizados en la función de recompensa que es común para los dos enfoques.

```
public override void OnActionReceived(float[] vectorAction)
{

    if (Mathf.FloorToInt(vectorAction[0]) == 0)
    {
        nCuraciones += 1;
        nAcciones += 1;
        if (nodoActual.nVirus == 1)
        {
            nodoActual.nVirus = 0;
            AddReward(0.01f);
        }
        else if (nodoActual.nVirus == 2)
        {
            nodoActual.nVirus = 0;
            AddReward(0.1f);
        }
        else if (nodoActual.nVirus == 3)
        {
            nodoActual.nVirus = 0;
            AddReward(1f);
        }
    }
    else
    {

        Nodo nuevoNodo = tablero.mapa.GetNodos()[Mathf.FloorToInt(vectorAction[0]) - 1];
        var distancias = tablero.mapa.DistanciasNodo(nodoActual.GetId());
        var distanciaANuevoNodo = distancias[(int)nuevoNodo.GetId()];

        nodoActual = nuevoNodo;
        AddReward(-0.1f * distanciaANuevoNodo);
        speed = Vector2.Distance(this.transform.position, nodoActual.transform.position);

        nMovimientos += distanciaANuevoNodo;
        nAcciones += distanciaANuevoNodo;
    }
}
```

FIGURA 36. FUNCIÓN DE RECOMPENSA DEL ENTRENAMIENTO DE DECISIÓN DEL AGENTE PARA AMBOS ENFOQUES.

Dependiendo de la acción escogida después de enmascararlas, se realiza un proceso u otro. En el caso de que el agente haya escogido el índice cero del vector de acción, quiere decir que ha escogido la acción de curar en el nodo en el que se encuentre. Por consiguiente, se contabiliza

una curación, una acción y se le otorga una recompensa dependiendo del número de virus que se encontrasen en el nodo en el momento de curar. La recompensa varia de una manera exponencial, siendo un nodo con tres virus el que mayor recompensa otorga al agente.
Si por otra parte el agente decide cualquier otro valor de acción, quiere decir que ha decidido desplazarse al nodo. En cuyo caso, se selecciona el nodo escogido y se calcula la distancia desde el actual al destino. Se le aplica un castigo proporcional a la distancia que va a recorrer o número de saltos que va a hacer, y se le desplaza al nodo objetivo contabilizando tanto los movimiento que hace como las acciones usadas para llegar hasta él.

También al igual que en el experimento anterior de curación máxima, la restricción de las cuatro acciones máximas por turno queda eliminada, permitiendo al agente poder curar todos los virus existentes en el mapa o, en su defecto, la mayoría de los nodos que se le permitan.

Teniendo listas la funciones, se ha pasado a la configuración de los parámetros del componente "Behavior Parameters" para la política del agente. Dependiendo del enfoque, se ha configurado de diferente manera. Para el enfoque de movimiento local se ha establecido de la siguiente forma:

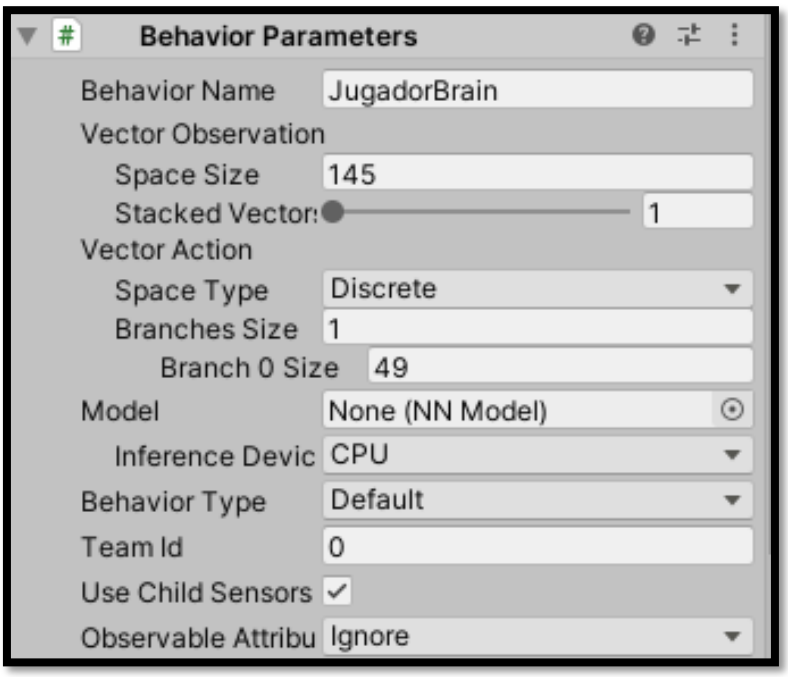


FIGURA 37. COMPONENTE "BEHAVIOR PARAMETERS" DEL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (ENFOQUE MOVIMIENTO LOCAL).

- Vector Observation
    - Space size. En este caso se ha establecido un tamaño de 145. Que, siguiendo la información recogida con la función “CollectObservations ()” citada anteriormente, corresponde con tres vectores de un tamaño de 48 (nodos totales) a uno por cada tipo de información (nodo actual del agente, nodos adyacentes y nodos con virus). Esto haría un total del espacio de 144, que, sumando la información del número de virus en el nodo actual del agente, sumaría 145.
    - Stacked Vector. Al igual que en los otros experimentos, establecido a uno. Es decir, solo se usa la información de la iteración del entrenamiento actual.
- Vector Action
    - Space type. De la misma forma, se ha seleccionado un espacio de tipo discreto.
    - Branches size. Para este experimento hay que añadir la posibilidad de la acción curar, por lo que esta acción más los movimientos a alguno de los 48 nodos se corresponde a un total de 49 acciones a enmascarar.

Sin embargo, tratándose del enfoque por movimiento global, el componente se ha configurado como puede verse a continuación:

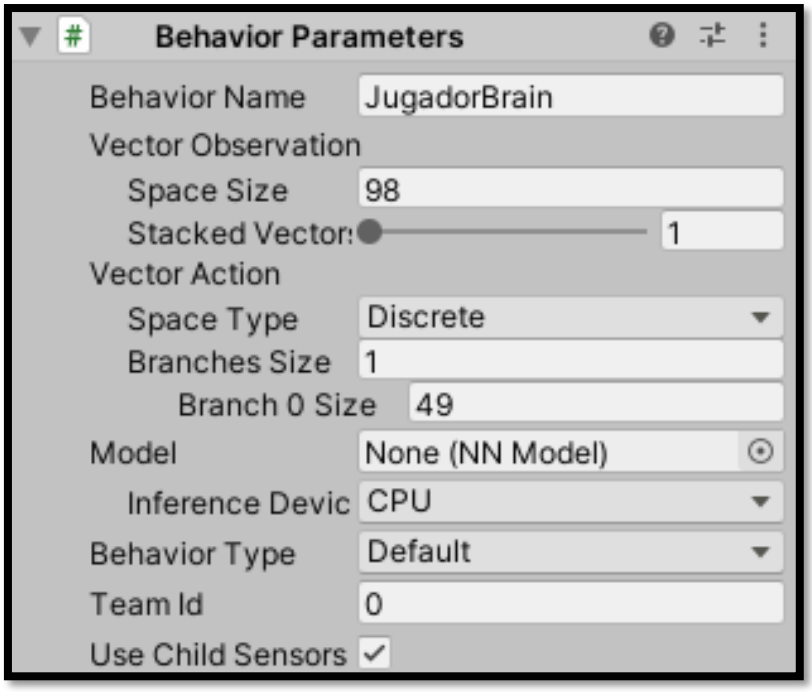


FIGURA 38. COMPONENTE “BEHAVIOR PARAMETERS” DEL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (ENFOQUE MOVIMIENTO GLOBAL).

- Vector Observation
    - Space size. En este caso se ha establecido un tamaño de 49. Que, siguiendo la información recogida con la función "CollectObservations ()" citada anteriormente para el movimiento global, corresponde con dos vectores de un tamaño de 48 (nodos totales) cada uno por tipo de información (distancias a cada uno de los nodos y virus en cada nodo). Esto haría un total del espacio de 96, que, sumando la información del número de virus en el nodo actual del y el número de acciones que el quedan al agente, sumaría 145.
    - Stacked Vector. Al igual que en los otros experimentos, establecido a uno. Es decir, solo se usa la información de la iteración del entrenamiento actual.
- Vector Action
    - Space type. De la misma forma, se ha seleccionado un espacio de tipo discreto.
    - Branches size. Para este experimento hay que añadir la posibilidad de la acción curar, por lo que esta acción más los movimientos a alguno de los 48 nodos se corresponde a un total de 49 acciones a enmascarar.

En cuanto al intervalo de decisión, esta vez se le ha llamado al agente manualmente cuando tenía que tomar una elección. Para ello, se ha usado la función de la librería "RequestDecision ()" llamándola bajo las condiciones que pueden verse en la siguiente Figura 39:

```
private void FixedUpdate()
{
    if (nodoActual != null)
    {

        this.transform.position = Vector2.MoveTowards(this.transform.position, nodoActual.transform.position, speed * Time.deltaTime);
        if (Vector2.Distance(this.transform.position, nodoActual.transform.position) == 0)
        {
            if (tablero.mapa.SinVirus())
            {
                EndEpisode();
            }
            if (nAcciones >= nAccionesMaximas)
            {
                EndEpisode();
            }
            else
            {
                this.RequestDecision();
            }
        }
    }
}
```

FIGURA 39. INTERVALO DE DECISIÓN DEL ENTRENAMIENTO DEL AGENTE.

Cada segundo del programa se comprueba si el agente ha terminado de curar o si se le han agotado las acciones disponibles, en cuyo caso se termina la iteración del entrenamiento. De no

ser así, se ha llamado a la función para que continúe tomando decisiones. Ambos enfoques implementan esta misma función.

Una vez se ha configurado todo lo necesario para el entrenamiento, se ha pasado a ejecutar el aprendizaje del agente. Realizado el entrenamiento, se ha comparado el resultado de los dos enfoques que se han planteado. Siendo el enfoque por movimiento local en color gris y el realizado por movimiento global en naranja.

Como en los anteriores experimentos, se han usado las siguientes gráficas de TensorBoard para ver los datos obtenidos.

- Cumulative Reward. Representa la recompensa obtenida por el agente a lo largo del entrenamiento realizado por ambos enfoques. Como se puede ver en la gráfica comparativa, ambos consiguen un resultado prácticamente idéntico. Consiguiendo el movimiento global estabilizarse algunas iteraciones antes que el local. Ambos alcanzan una recompensa de 0.7022 y 7.083 respectivamente.

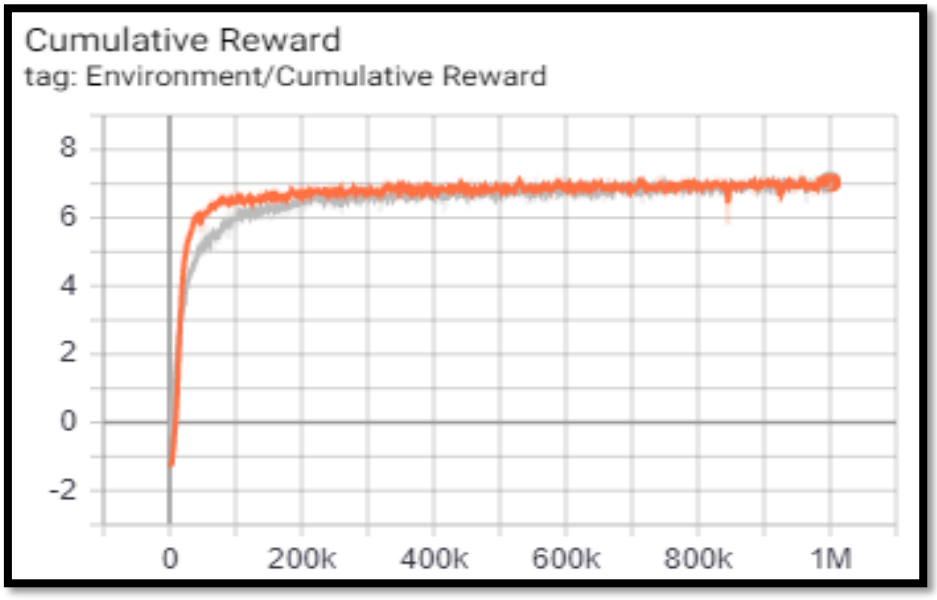


FIGURA 40. GRÁFICA "CUMULATIVE REWARD" PARA EL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (COMPARATIVA DE ENFOQUES).

- Entropy. Puede observarse que la aleatoriedad con la que el agente toma decisiones es alta al comienzo del entrenamiento para ambos enfoques, pero en seguida va

descendiendo y estabilizándose el valor a partir de la mitad del entrenamiento. Finaliza con un valor de 0.2108 para el movimiento global y de 0.2484 para el local.

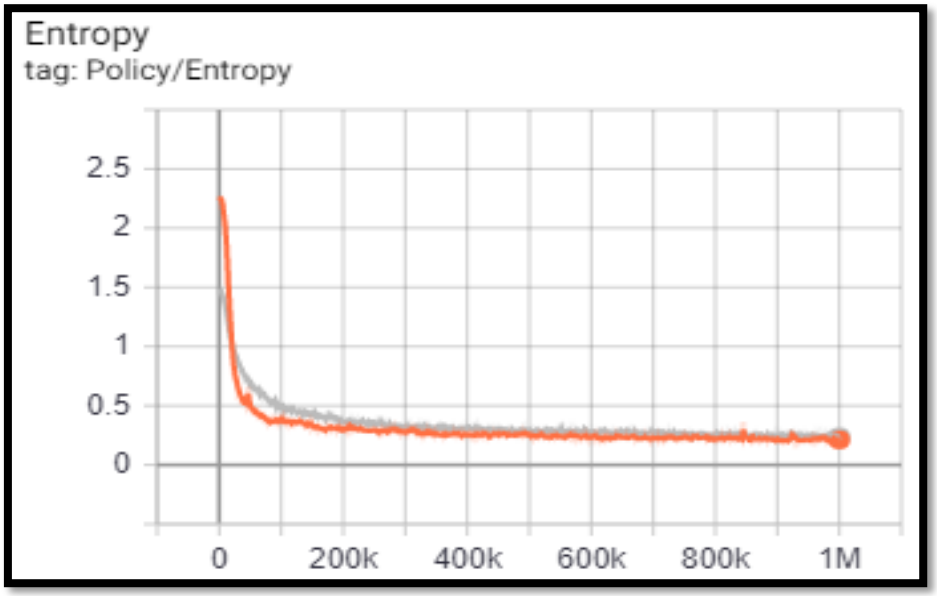


FIGURA 41. GRÁFICA "ENTROPY" PARA EL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (COMPARATIVA DE ENFOQUES).

- Policy Loss. En el caso de este experimento, puede verse como los valores de los dos enfoques van oscilando prácticamente todo el entrenamiento, lo que indica cambios en las políticas continuamente. Sin embargo, a medida que avanzan en las iteraciones el intervalo de los valores de oscilación disminuyen en el enfoque local, no siendo así en el global que posee algunos externos. Así pues, el cambio de las políticas sería despreciable con un valor de 0.1377 en el enfoque local y de 0.1344 en el global, aunque con algunos casos extremos aislados.

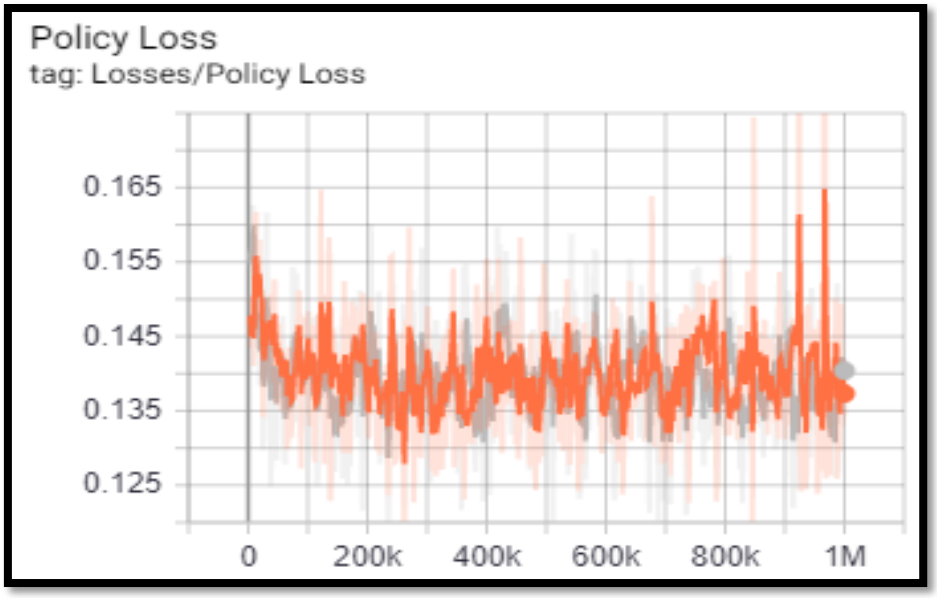


FIGURA 42. GRÁFICA "POLICY LOSS" PARA EL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (COMPARATIVA DE ENFOQUES).

- Value Loss. Observando la gráfica de predicción de recompensa, se comienza con un valor de 0.405 para el movimiento local y de 9.998 para el global que rápidamente comienzan a descender en los primeros pasos del entrenamiento. Esto es, que el agente alcanza rápidamente el valor deseado de predicciones satisfactorias de la recompensa futura. Alcanzando el final del entrenamiento ambos se estabilizan finalizando con un valor de 0.0258 y 0.01599 respectivamente. En la Figura 44 también se puede ver como en el enfoque global se alcanza un valor más bajo, pero de forma más inestable en algunos puntos distinguidos.

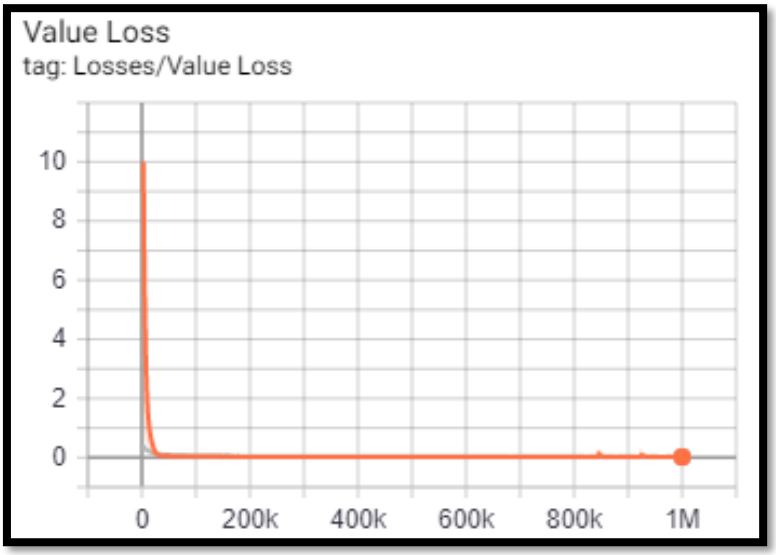


FIGURA 43. GRÁFICA 1 "VALUE LOSS" PARA EL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (COMPARATIVA DE ENFOQUES).

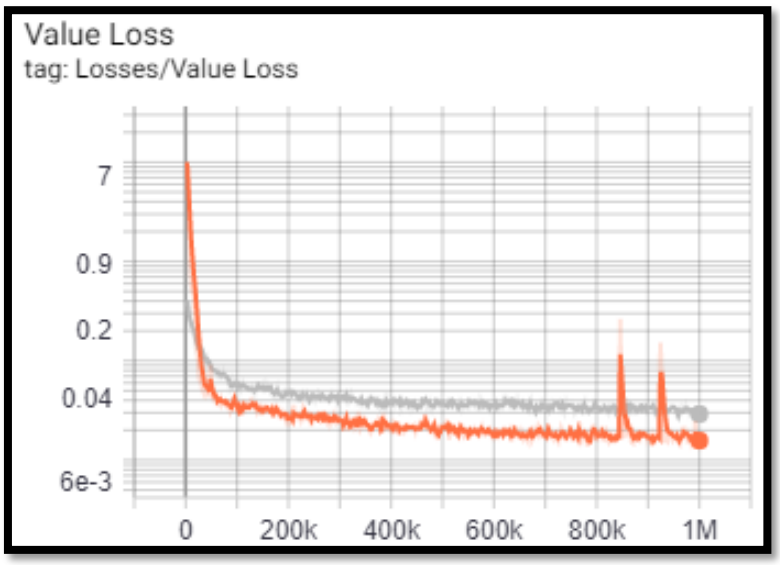


FIGURA 44. GRÁFICA 2 "VALUE LOSS" PARA EL ENTRENAMIENTO DE DECISIÓN DEL AGENTE (COMPARATIVA DE ENFOQUES).

En resumen, de la comparativa de los enfoques, con el uso de ambos métodos se le hace posible al agente aprender a decidir entre las acciones de curar o mover. Realizando el aprendizaje de la manera más óptima de terminar con los virus en el mapa. Aprendiendo a curar a la vez que decide su próximo movimiento.

Problemas y dificultades

A parte de lo que se ha mencionado a lo largo del experimento, en un principio se planteó el uso de dos ramas de acciones para realizar la toma de decisión del agente. Es decir, una rama para la curación y otra para los movimientos.

Al utilizar este método, no se ha podido completar ninguno de los entrenamientos intentados. Esto se debe a que el agente, no decide sobre una de las dos ramas, si no que aplica ambas ramas al mismo tiempo, por lo que no es aplicable para la meta que se buscaba con el experimento.

El otro problema ya mencionado, era el movimiento del agente por el camino más corto entre nodos, que como se ha explicado se puede resolver mediante el aprendizaje del movimiento primero, o desarrollando el movimiento manualmente y dejando la información acerca de él al agente.

Comparativa

Para concluir el desarrollo y experimentos del presente proyecto, se ha realizado una programación y comparativa del agente entrenado frente a otros agentes que funcionen con otros algoritmos. En este caso, se han hecho prueba frente a un agente con el algoritmo voraz y otro con un comportamiento aleatorio. Todos ellos se han puesto en una partida para ver la eficiencia y la ayuda que pudieran prestar a un jugador humano en términos de alargar la partida.

Para la prueba, se ha hecho uso de la función “Heuristic ()” que se ha sobre escrito para programar un comportamiento del agente según los algoritmos citados.

En el caso del voraz, como dicta el algoritmo, el agente curará primero los nodos con mayor recompensa en cada iteración del entorno. Por ello, se ha recogido información de todos los

nodos con virus ordenados por número de virus y por cercanía al agente. Y guardada la información ordenada, se le ha programado al agente ir a cada nodo específico y curarlo en caso de haber llegado a él. Para el movimiento, se ha programado una función de búsqueda de caminos basada en el algoritmo de Dijkstra.

En cuanto al algoritmo aleatorio, simplemente se ha recogido la información de los nodos con virus y una vez guardada, se escoge un nodo aleatorio al que ir a curar.

Las pruebas de los algoritmos con los que funciona el agente se han realizado sobre un escenario similar a una partida real, siguiendo las normas adaptadas comentadas en la introducción del desarrollo. En consecuencia, se han podido tomar medidas para dictar cuál es el agente que mejor rendimiento proporcionaría en la partida.

Las medidas utilizadas para realizar la comparativa han sido las siguientes:

- Número de movimientos. Mide la cantidad de movimientos realizados por el agente a lo largo de las partidas.
- Número de curación. Mide la cantidad total de curaciones que se han logrado a lo largo del total de partidas.
- Número de turnos. Turnos del jugador transcurridos en el total de partidas probadas.
- Número de partidas. Partidas totales probadas con cada agente.

Establecidas las medidas se ha pasado a realizar las pruebas. Se han obtenido los resultados que pueden verse en la siguiente Tabla 1. Resultados de los algoritmos.:

	N.º de Curaciones	**N.º de Movimientos**	**N.º de Turnos**	**N.º de Partidas**
IA Entrenada	3835	7386	2805	250
IA Voraz	2834	9267	3025	250
IA Aleatoria	240	8873	2278	250

TABLA 1. RESULTADOS DE LOS ALGORITMOS.

A la vista de los resultados, se puede comprobar como el agente con el algoritmo aleatorio queda prácticamente fuera de la comparativa atendiendo al número de curaciones y de turnos

conseguidos en el total de las partidas. Por lo que el agente con más rendimiento estaría entre el algoritmo voraz y el agente entrenado con los experimentos.

Tratando el número de turnos, la diferencia entre los dos no es muy notoria. Aun así, el algoritmo voraz es superior, por lo que en términos de eficiencia y rendimiento este algoritmo en un escenario real alargaría más el tiempo de juego de la partida. Lo que, en términos del juego, supondría mayor tiempo y, por tanto, probabilidad para poder ganarla.

Esto se debe a la naturaleza del algoritmo. Como se ha explicado en la introducción del desarrollo de las reglas del juego, si los brotes llegan a un máximo de ocho la partida se da por terminada. Debido al algoritmo voraz buscando siempre la mayor recompensa, es decir, los nodos con tres virus, se reduce la probabilidad de que surjan brotes durante la partida. No siendo necesario realizar muchas curaciones y alargando la partida de forma implícita.

No obstante, el agente entrenado no dista mucho en turnos totales. Además, en cuanto al número de curaciones, en el caso del agente entrenado el valor es mucho mayor que el algoritmo voraz. Es decir, en un escenario de juego el algoritmo entrenado obtendría por curar una mayor recompensa que el voraz, al mismo tiempo siendo posible también alargar la partida lo máximo.

Conclusiones

A lo largo del primer experimento se ha conseguido que el agente entrenado a través del aprendizaje por refuerzo sea capaz encontrar el camino más corto al nodo de destino por decisión propia. Ya sea tanto en distancias de menor medida a nodos adyacentes como en largas distancias. En el caso de las distancias largas, ha sido necesario usar el entrenamiento por Curriculum, ya que sin este no se ha logrado que el agente aprenda. Puesto que, al encontrarse la única recompensa del escenario a mucha distancia, el agente se quedaba con facilidad realizando saltos en los nodos adyacentes no llegando nunca al nodo de destino, y, por consiguiente, a obtener la recompensa.

En el segundo experimento se ha logrado que el agente cure los nodos en los que aparecen virus de una manera óptima que le llevase a la mayor recompensa. En los primeros intentos el entrenamiento no resultó exitoso porque, al igual que en el primer experimento, en un escenario de prueba con escasas ciudades infectadas con virus, el agente no lograba alcanzar estas ciudades mediante movimientos aleatorios para efectuar una curación que le generase recompensa. Para solucionarlo resultó necesario realizar el entrenamiento por Curriculum, dejando las primeras fases del entrenamiento para que el agente aprenda a llegar a los nodos por el camino más corto con rutas de longitud creciente. Otra de las alternativas posibles, fruto del conocimiento adquirido con el tercer experimento, hubiese sido aumentar la cantidad de ciudades infectadas inicialmente para que el agente tuviese más probabilidad de encontrarlas mediante movimientos aleatorios y obtener recompensas, aprendiendo de forma simultánea a moverse entre ciudades siguiendo el camino más corto y a visitar en primer lugar las ciudades con más virus.

Como tercer experimento, se ha realizado una combinación de lo aprendido a lo largo de los otros experimentos, probando además otro enfoque alternativo con relación al movimiento del agente para evaluar su impacto en el proceso de aprendizaje. Uno de los enfoques utilizados ha sido el mismo que en experimentos anteriores, permitiendo al agente moverse solo a ciudades adyacentes y obligándole a aprender a moverse de forma óptima entre ciudades alejadas. El otro enfoque ha sido incorporar un algoritmo de cálculo de camino más corto y proporcionar

esta información al agente, para evitarle tener que aprender a moverse de forma óptima, permitiéndole moverse entre ciudades alejadas gastando tantas acciones como distancia haya entre ambas ciudades. Los resultados de ambos enfoques han sido prácticamente idénticos una vez finalizado el entrenamiento, resultando ambos exitosos para conseguir que el agente se moviese adecuadamente y curase las ciudades con mayor cantidad de virus. Además, se ha utilizado un escenario de entrenamiento con mayor cantidad de virus, comparado con el segundo experimento, para asemejarse más al escenario final del juego. Este cambio ha permitido ver que, con mayor probabilidad de conseguir recompensas, el entrenamiento por Curriculum dejaba de ser necesario.

Por otro lado, se ha comparado el rendimiento en una partida del agente entrenado frente a otros algoritmos de IA. La comparación se ha realizado con el algoritmo voraz, uno aleatorio y el algoritmo desarrollado. Atendiendo a los resultados obtenidos, el algoritmo desarrollado se ha situado muy por encima del aleatorio y prácticamente a la par que el voraz, superándolo en ocasiones. Sin embargo, no se ha conseguido realizar un entrenamiento en el entorno de una partida normal, es decir, con la información respecto al mazo de cartas, control de turnos, etc. Puesto que al ser una información mucho más compleja que la de entornos simulados, no ha sido posible proporcionársela al agente de una forma entendible y usable para su entrenamiento.

La conclusión de todos estos experimentos e investigaciones es que, si bien es posible aplicar el aprendizaje por refuerzo sobre un agente en un entorno de juego de mesa y conseguir que este aprenda a jugar de una forma relativamente buena, existen numerosos retos y dificultades a superar para llevar a cabo un entrenamiento exitoso, y, pese a todo, el agente entrenado no tiene por qué ser superior a otros algoritmos más simples, como los voraces, que buscan maximizar la recompensa obtenida en base a una heurística proporcionada por el programador.

Limitaciones de los experimentos

Una vez se ha investigado y realizado los experimentos, se ha podido comprobar como principal limitación que debido a la complejidad de las reglas de juego y la complejidad que supone convertir el estado de juego completo de “Pandemic” a un formato consumible por el algoritmo de aprendizaje automático, no se ha podido entrenar sobre el juego real. Esto hace que el agente

inteligente no pueda tener en consideración todo el estado del juego a la hora de tomar decisiones y, por tanto, su rendimiento esté limitado al estar actuando con información incompleta, lo que en la práctica supone que su rendimiento no pueda llegar a ser superior al del algoritmo voraz al intentar alargar el estado de la partida retrasando la condición de derrota.

Trabajo Futuro

Terminado el desarrollo y los experimentos se han planteado algunas posibles extensiones y nuevas vías de desarrollo para el presente proyecto. Entre ellas, siendo las más destacables las siguientes:

- Implementación del resto de acciones y mecánicas. Una posible mejora sería extender las reglas asemejándose más a las originales del juego. Esto llevaría a una implementación de las demás acciones del juego disponibles para el jugador. Entre ellas, la acción de investigación de los virus, que con un entrenamiento del agente satisfactorio llevaría a un escenario de la partida donde sería posible ganar. Sumando así de forma procedural, tanto la implementación de reglas con sus respectivas acciones, ajustándose aún más a los límites del motor y de la librería.

- Entrenamiento de varios agentes. Como se ha explicado en párrafos anteriores de la memoria, la librería de ML-Agents permite un entrenamiento basado en multi agente. Esto permite el entrenamiento en juegos competitivos de un agente en aprendizaje frente a sí mismo, aprendiendo entre ellos. Siendo “Pandemic” un juego de naturaleza cooperativa, se podría investigar formas de aplicar este tipo de entrenamiento para que dos o más agentes cooperen entre sí compartiendo información de la partida. Asemejándose más al juego original.

- Cooperación entre jugador humano e IA. De una manera similar a la mejora anterior sobre el proyecto, podría extenderse el trabajo tratando de entrenar al agente mediante una cooperación con un jugador humano. Pasando la información de las acciones tomadas por el humano al agente, y que en base a estas tome las decisiones oportunas para completar un entrenamiento satisfactorio.

Anexo

Pliego de Condiciones

Características técnicas de equipo

Para la realización del presente proyecto, ha sido necesario el uso de un equipo con un hardware robusto por los requisitos necesarios para realizar el entrenamiento de las redes neuronales. Por ello, se ha utilizado un equipo de sobremesa con los siguientes componentes hardware:

- CPU: AMD Ryzen 7 3800X 8-Core Processor (3.9GHz)
- Placa Base: GIGABYTE X570 AORUS ELITE
- RAM: Kingston HyperX Fury RGB 32GB DDR4 3200Mhz (2x16Gb)
- GPU: NVIDIA Evga GeForce RTX 2070 XC Gaming

Además de los componentes hardware se han utilizado los siguientes componentes software:

- Microsoft Windows 10 Professional N (x64) Build 19041.450
- Microsoft Office 365 ProPlus
- Unity 2019.4.0f1 Personal Version
- ML-Agents 0.17.0
- Install Python 3.7.6
- TensorFlow 2.0.1

Manual de Usuario

En este apartado se va a detallar tanto los pasos como las versiones de los programas necesarios para la ejecución de las diferentes pruebas realizadas a lo largo del proyecto.

Preparación del entorno de entrenamiento

Como se ha citado en el apartado anterior, los programas usados en el proyecto y necesarios para las pruebas del proyecto son: Unity en su versión 2019.4.0f1, librería ML-Agents en su versión 0.17.0 (Release 3), Python en su versión 3.6.1 o superior (usada la 3.7.6) y la herramienta TensorFlow en su versión 2.0.1.

Para realizar la instalación de Unity Personal, basta con descargar Unity Hub de la página principal de Unity que puede encontrarse en https://store.unity.com/download?ref=personal.

Una vez instalado, se accede a la página https://unity3d.com/get-unity/download/archive para descargar e instalar a través de Unity Hub la versión necesaria para el proyecto (2019.4.0). Instalada la versión, se selecciona el botón "Add" en la pestaña de "Projects", y se selecciona la carpeta raíz del proyecto para añadirlo a Unity como puede verse en la siguiente Figura 45.Agregar proyecto en Unity Hub.:

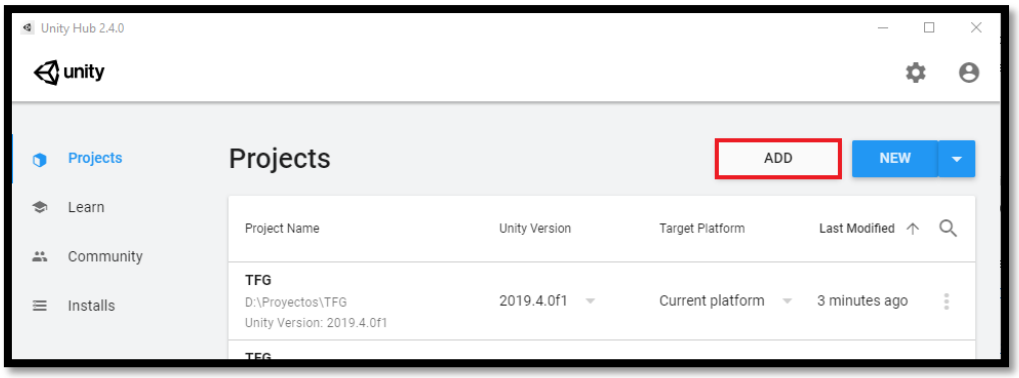


FIGURA 45.AGREGAR PROYECTO EN UNITY HUB.

El siguiente paso es instalar la librería ML-Agents. Para ello es necesario abrir el proyecto agregado anteriormente y en caso de no instalarse las dependencias automáticamente, se deberá ir a la pestaña "Window -> Package Manager" y buscar la librería en la lista. A continuación, se muestran las Figura 46 y Figura 47 con los pasos a seguir.

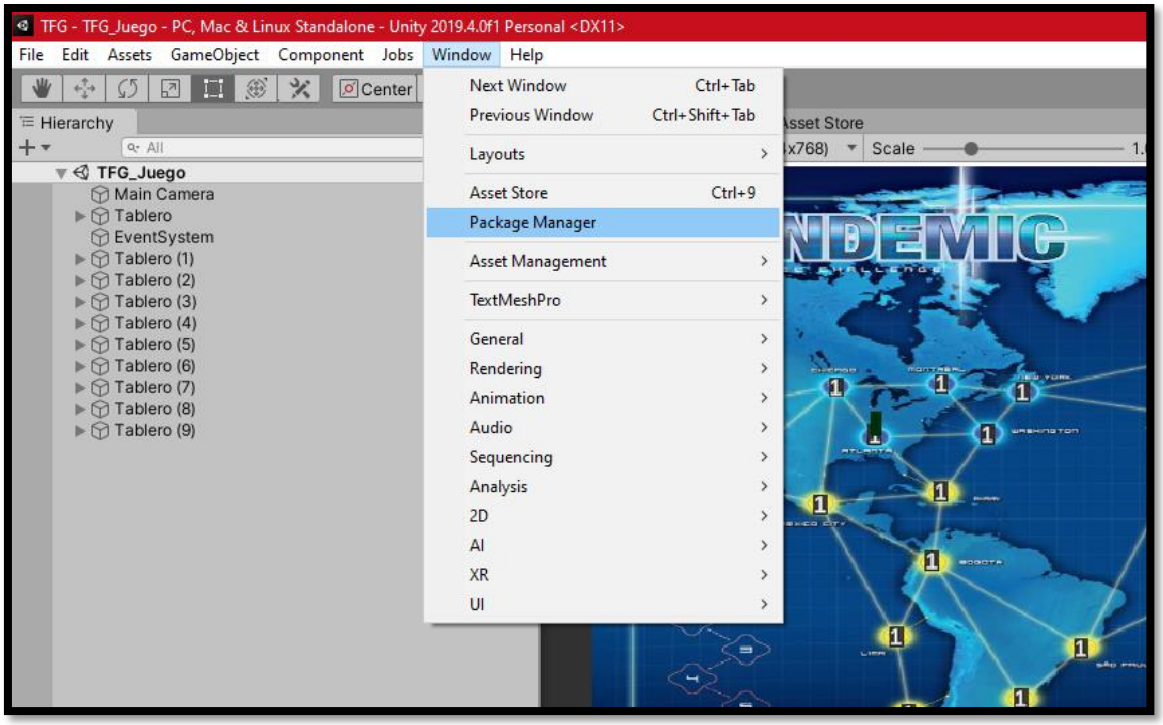


FIGURA 46. VISUALIZAR LAS LIBRERÍAS DISPONIBLES.

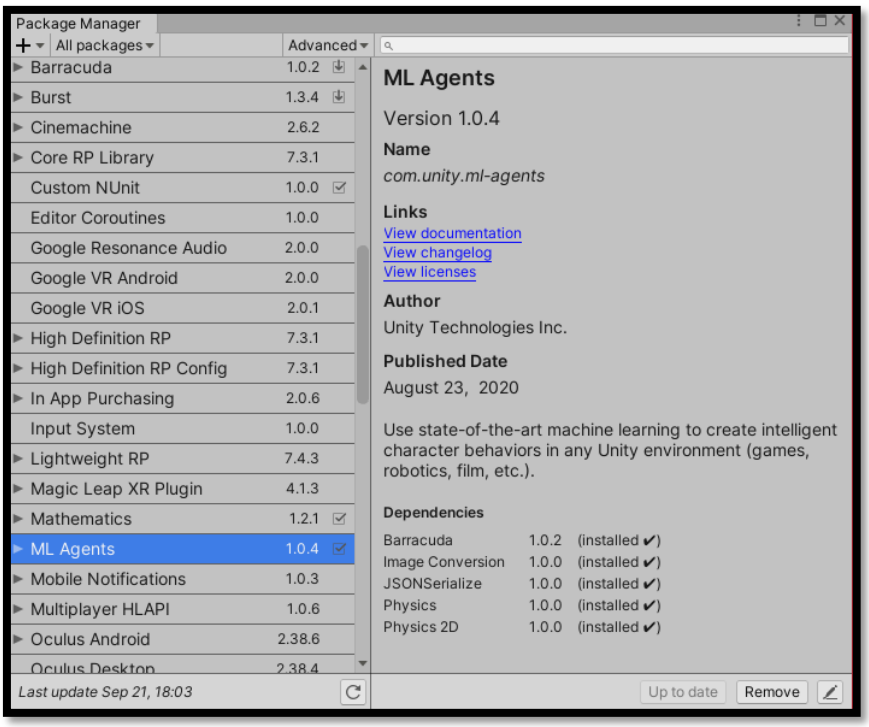


FIGURA 47. LIBRERÍA ML-AGENTS EN LA LISTA DE "ALL PACKAGES".

Por último, es necesario instalar Python y TensorFlow. Para instalar Python se puede realizar de manera sencilla descargando e instalando la versión más reciente desde la página web https://www.python.org/downloads/. Con ella, se instalará también la herramienta "pip3" para gestionar los paquetes de Python. Hecha la instalación solo quedaría instalar el paquete de Python "mlagents", que instalará consigo la dependencia TensorFlow y que puede realizarse ejecutando desde una línea de comandos la sentencia: "pip3 install mlagents ".

Entrenamiento del agente

Con las herramientas ya instaladas siguiendo los pasos del apartado anterior, se tendría listo el entorno para ejecutar el juego y el agente desarrollado a lo largo del trabajo. En el caso que se quiera realizar un nuevo entrenamiento para el agente, se deberán seguir las siguientes pautas.

Primero, desde la ventana de las carpetas del proyecto, se deberá abrir dentro de la carpeta "Scenes" una de las tres escenas de Unity correspondientes a cada uno de los experimento desarrollados. Véase en la siguiente Figura 48.

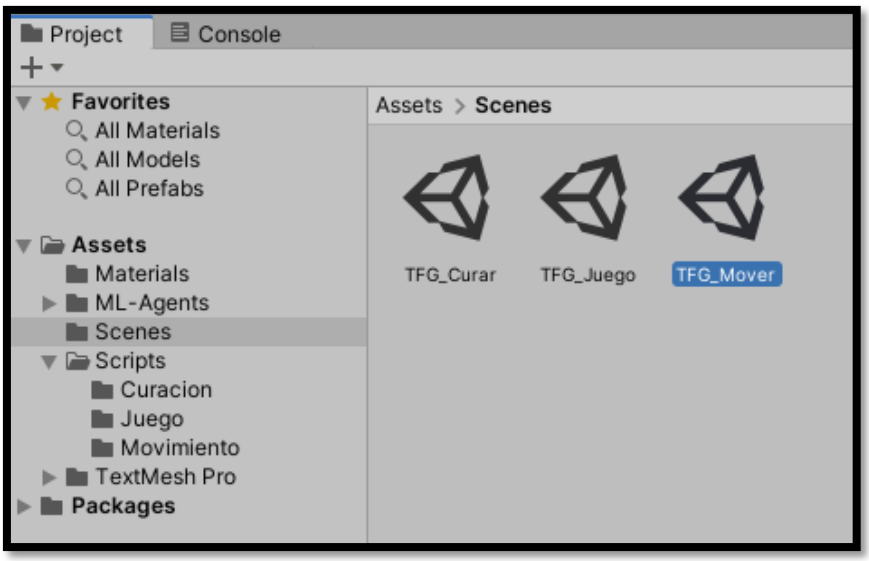


FIGURA 48. ESCENAS DEL PROYECTO.

Una vez seleccionada la escena, para empezar el entrenamiento es necesario abrir una ventana de comandos y escribir la siguiente instrucción:

"mlagents-learn config/ppo/TFG_curriculum_mover.yaml --run-id=NombreDeDestino".

Siendo el segundo argumento el fichero “Yaml” de configuración con los parámetros de entrenamiento y la opción “--run-id” para dar nombre a la carpeta donde se guardarán los resultados. Introducida la instrucción el programa se quedará en espera como en la siguiente Figura 49.

FIGURA 49. PANTALLA DE LA CONSOLA DE COMANDOS EN ESPERA.

Para finalizar, solo quedaría ejecutar la escena desde Unity dándole al botón de “Play” que se muestra en la Figura 50 y esperar a que finalice el entrenamiento.

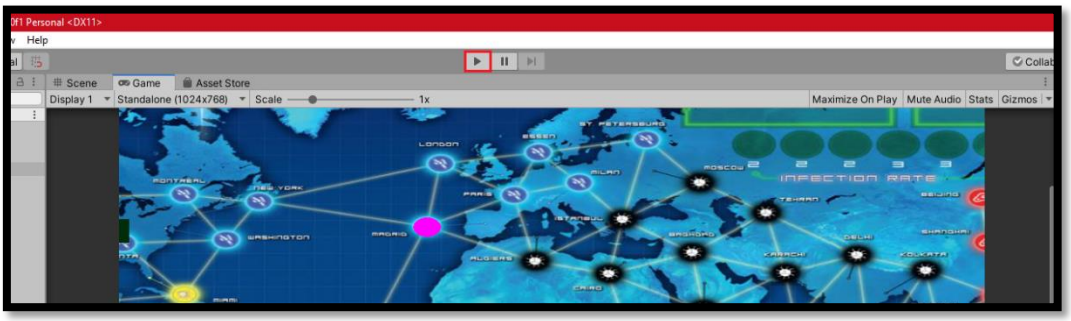


FIGURA 50. BOTÓN “PLAY” PARA EJECUTAR LA ESCENA.

Pruebas de las redes neuronales

Realizado el entrenamiento de una red, es posible asignarla el agente para probarlo en el entorno de juego. Los pasos por seguir para probar las redes neuronales son los siguientes dependiendo si se hace uso de las redes ya entrenadas proporcionadas con el proyecto o de una nueva red entrenada.

- Si se hace uso de las redes ya entrenadas durante los experimentos del presente trabajo, solo es necesario seleccionar la escena siguiendo los pasos del apartado anterior. Y, una vez seleccionada, se deberán desactivar primero todos los agentes menos uno, seleccionándolos desde la ventana “Hierarchy” de Unity. Tal y como se muestra en la siguiente Figura 51.

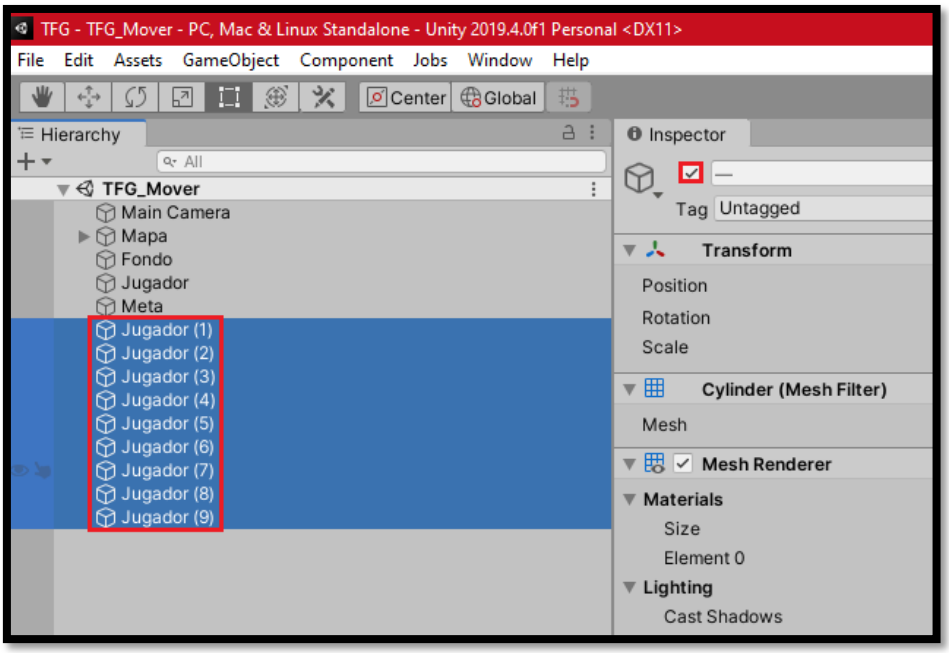


FIGURA 51. DESACTIVAR AGENTES DE LA ESCENA.

Después, se le asigna al agente restante el componente de la red neuronal correspondiente a la escena elegida dentro del campo “Model” del componente “Behavior Parameters” explicado en el apartado de desarrollo de la memoria. Estas se encuentran guardadas en la carpeta “Scripts” y dentro de la carpeta de cada experimento. En el caso de este ejemplo “Movimiento”. Quedando de la siguiente manera:

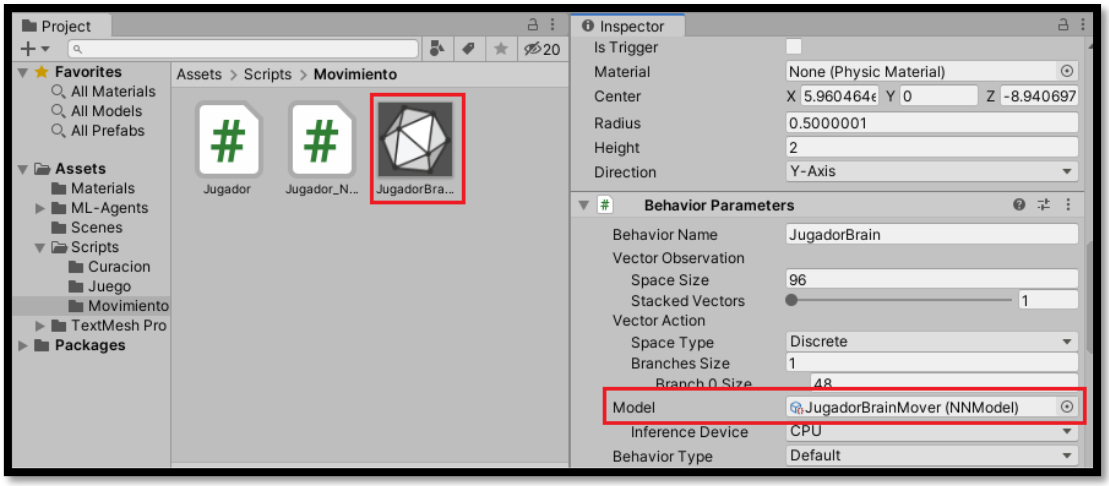


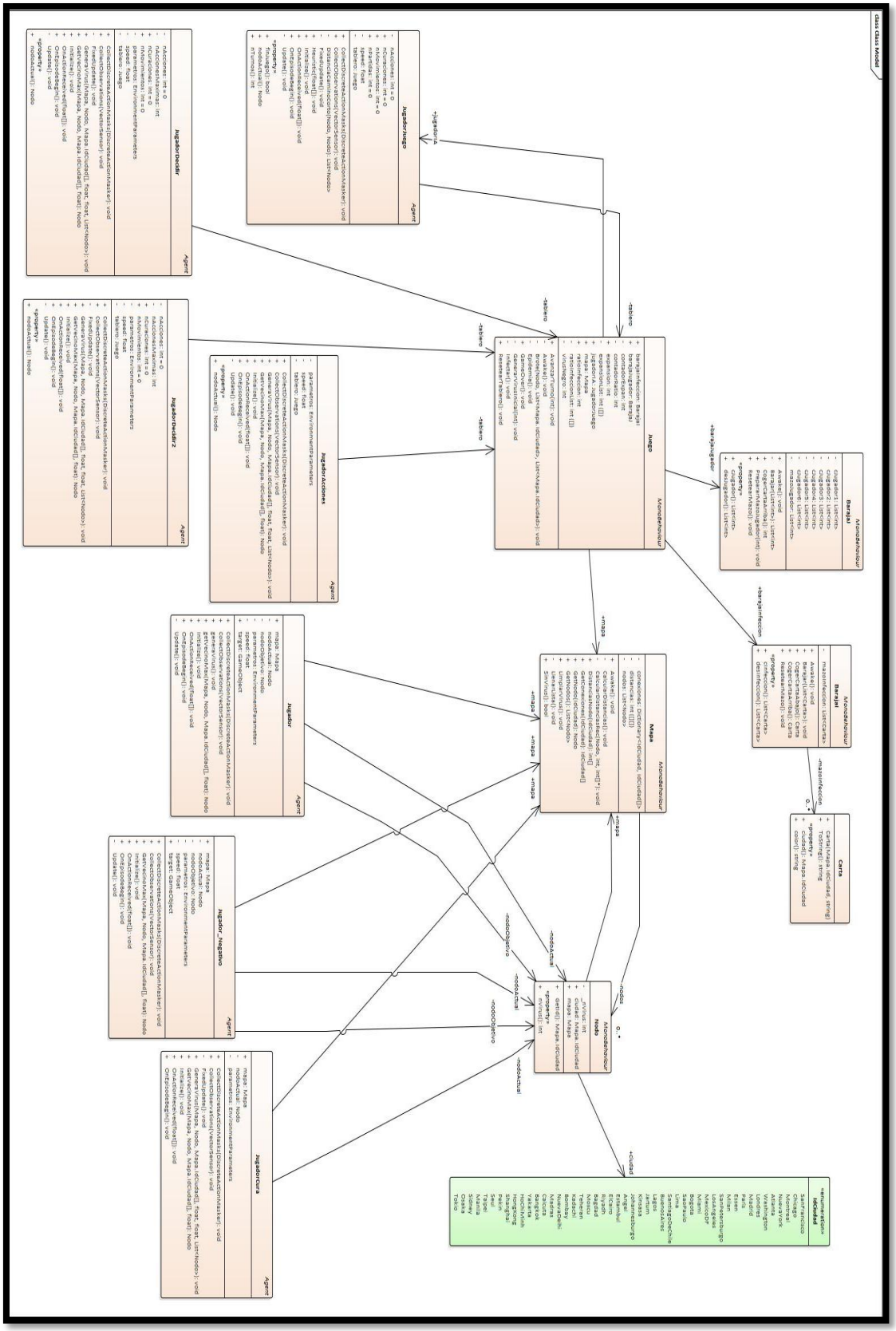
FIGURA 52. ASIGNAR LA RED NEURONAL AL AGENTE.

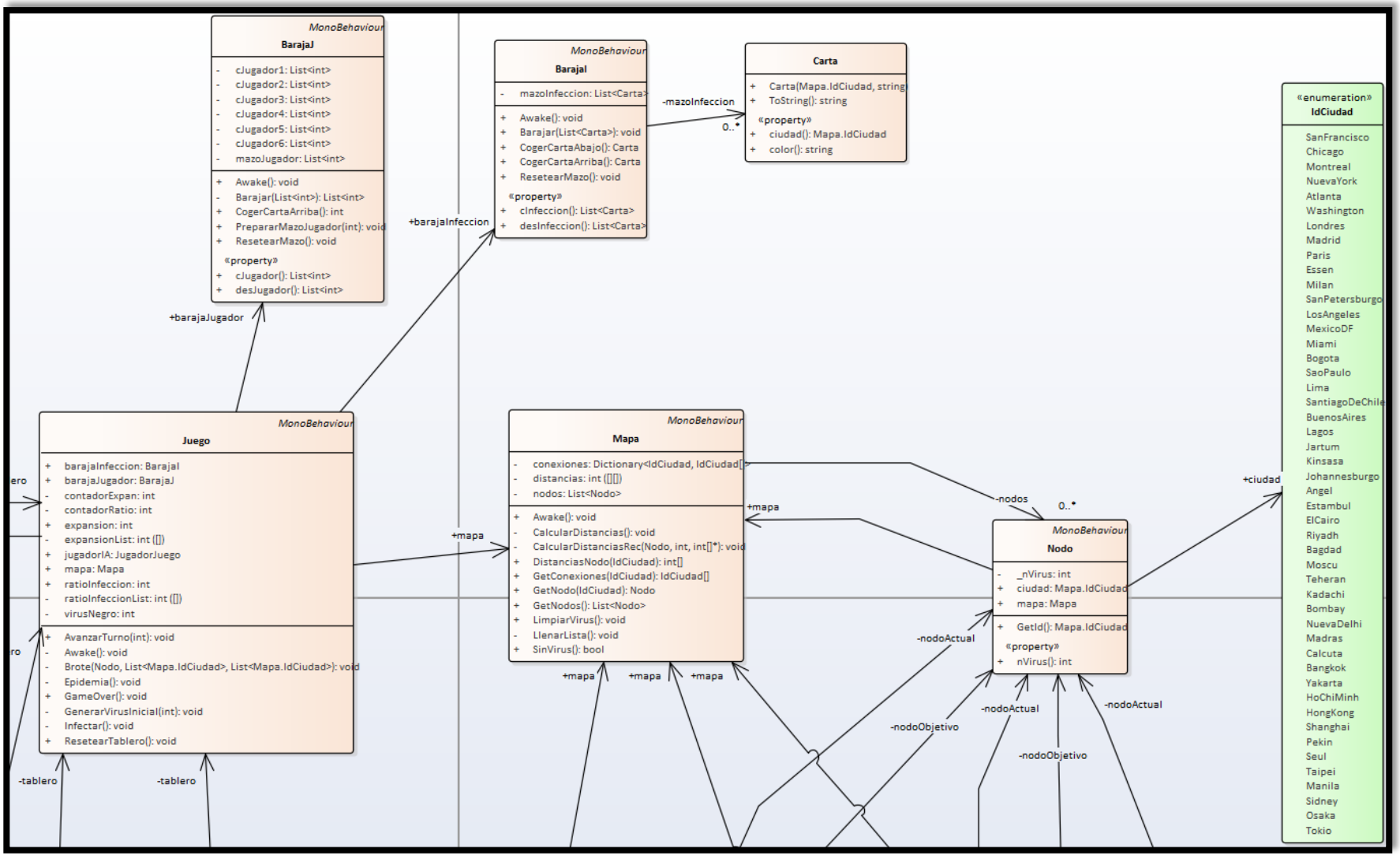
Por último, para empezar la prueba basta con darle al botón “Play” de la escena.

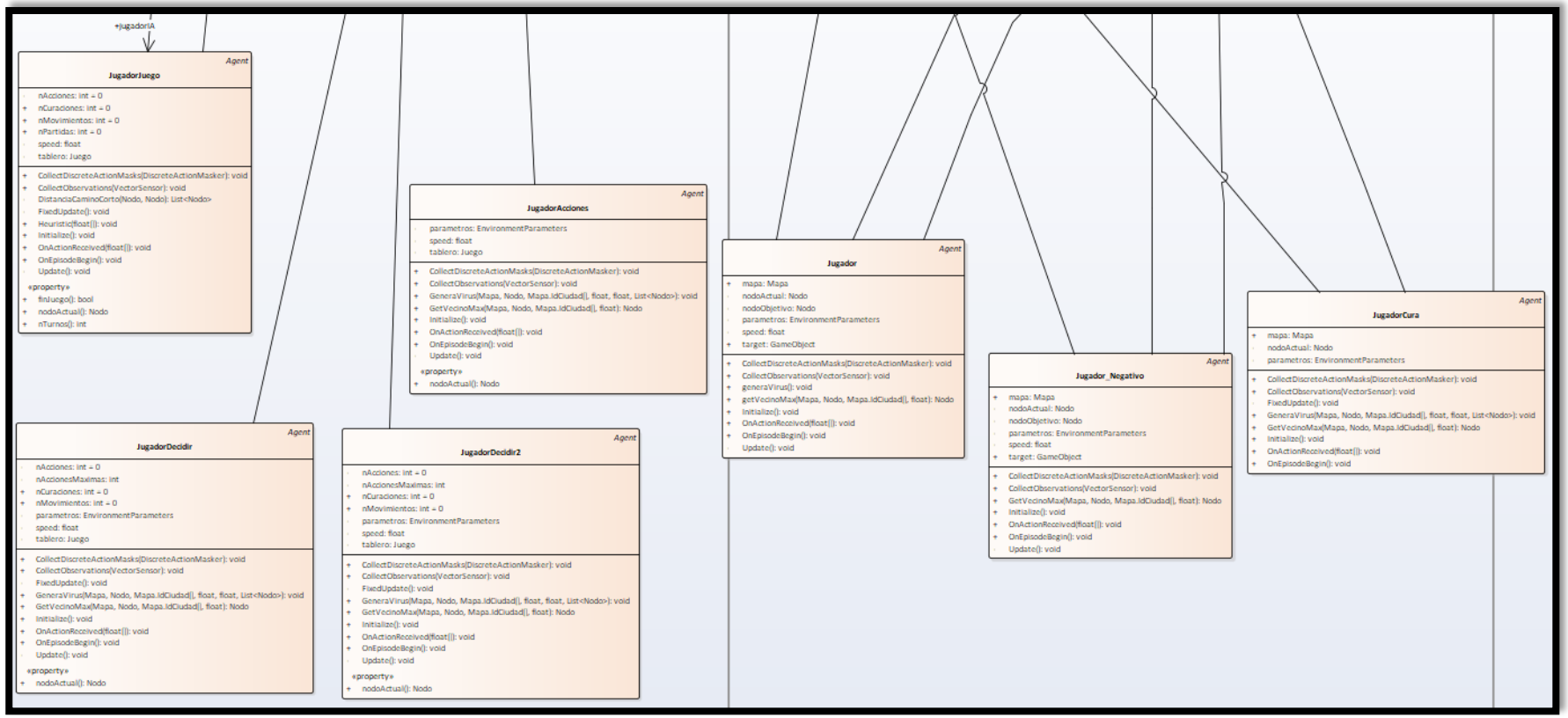
- Si se va a usar una nueva red entrenada, primero es necesario arrastrar a la ventana de los archivos de proyecto de Unity la red generada después del entrenamiento. Una vez se ha copiado dentro de la ventana “Project” se deben seguir los mismos pasos anteriores.

Diagrama de Clases

A continuación, se exponen los diferentes diagramas que componen las clases desarrolladas y programadas durante la realización del presente trabajo.


class Class Model
Agent
MonoBehaviour
Juego
Baraja1
Carta
Nodo
Mapa
idCiudad
Angel
Atlanta
Bangkok
Beijing
Bogota
Buenos Aires
Cairo
Chicago
Delhi
Essen
Estambul
Hong Kong
Jakarta
Johannesburgo
Lima
Londres
Los Angeles
Madrid
Manila
Mexico DF
Miami
Milan
Montreal
Moscu
Mumbai
Nueva Delhi
Nueva York
Paris
Riyadh
San Francisco
Santiago
Sao Paulo
Seul
Shanghai
Sidney
Taipei
Tehran
Tokio
Toronto


MonoBehaviour
BarajaJ
- cJugador1: List<int>
- cJugador2: List<int>
- cJugador3: List<int>
- cJugador4: List<int>
- cJugador5: List<int>
- cJugador6: List<int>
- mazoJugador: List<int>
+ Awake(): void
- Barajar(List<int>): List<int>
+ CogerCartaArriba(): int
+ PrepararMazoJugador(int): void
+ ResetearMazo(): void
«property»
+ cJugador(): List<int>
+ desJugador(): List<int>
MonoBehaviour
Barajal
- mazoInfeccion: List<Carta>
+ Awake(): void
+ Barajar(List<Carta>): void
+ CogerCartaAbajo(): Carta
+ CogerCartaArriba(): Carta
+ ResetearMazo(): void
«property»
+ cInfeccion(): List<Carta>
+ desInfeccion(): List<Carta>
Carta
+ Carta(Mapa.IdCiudad, string
+ ToString(): string
«property»
+ ciudad(): Mapa.IdCiudad
+ color(): string
«enumeration»
IdCiudad
SanFrancisco
Chicago
Montreal
NuevaYork
Atlanta
Washington
Londres
Madrid
Paris
Essen
Milan
SanPetersburgo
LosAngeles
MexicoDF
Miami
Bogota
SaoPaulo
Lima
SantiagoDeChile
BuenosAires
Lagos
Jartum
Kinsasa
Johannesburgo
Angel
Estambul
ElCairo
Riyadh
Bagdad
Moscu
Teheran
Kadachi
Bombay
NuevaDelhi
Madras
Calcuta
Bangkok
Yakarta
HoChiMinh
HongKong
Shanghai
Pekin
Seul
Taipei
Manila
Sidney
Osaka
Tokio
MonoBehaviour
Juego
+ barajaInfeccion: BarajaI
+ barajaJugador: BarajaJ
- contadorExpan: int
- contadorRatio: int
+ expansion: int
- expansionList: int ([])
+ jugadorIA: JugadorJuego
+ mapa: Mapa
+ ratioInfeccion: int
- ratioInfeccionList: int ([])
- virusNegro: int
+ AvanzarTurno(int): void
- Awake(): void
- Brote(Nodo, List<Mapa.IdCiudad>, List<Mapa.IdCiudad>): void
- Epidemia(): void
+ GameOver(): void
- GenerarVirusInicial(int): void
- Infectar(): void
+ ResetearTablero(): void
MonoBehaviour
Mapa
- conexiones: Dictionary<IdCiudad, IdCiudad[]>
- distancias: int ([][])
- nodos: List<Nodo>
+ Awake(): void
- CalcularDistancias(): void
- CalcularDistanciasRec(Nodo, int, int[]*): void
+ DistanciasNodo(IdCiudad): int[]
+ GetConexiones(IdCiudad): IdCiudad[]
+ GetNodo(IdCiudad): Nodo
+ GetNodos(): List<Nodo>
+ LimpiarVirus(): void
- LlenarLista(): void
+ SinVirus(): bool
MonoBehaviour
Nodo
- _nVirus: int
+ ciudad: Mapa.IdCiudad
+ mapa: Mapa
+ GetId(): Mapa.IdCiudad
«property»
+ nVirus(): int
+barajaInfeccion
-mazoInfeccion
0..*
+barajaJugador
+mapa
+mapa
-nodos
0..*
+ciudad
-nodoActual
-nodoActual
-nodoActual
-nodoObjetivo
-nodoObjetivo
-tablero
-tablero
+mapa
+mapa
+mapa


+jugadorIA
Agent
JugadorJuego
- nAcciones: int = 0
+ nCuraciones: int = 0
+ nMovimientos: int = 0
+ nPartidas: int = 0
- speed: float
- tablero: Juego
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
DistanciaCaminoCorto(Nodo, Nodo): List<Nodo>
FixedUpdate(): void
+ Heuristic(float[]): void
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Update(): void
«property»
+ finJuego(): bool
+ nodoActual(): Nodo
+ nTurnos(): int
Agent
JugadorAcciones
- parametros: EnvironmentParameters
- speed: float
- tablero: Juego
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
+ GeneraVirus(Mapa, Nodo, Mapa.IdCiudad[], float, float, List<Nodo>): void
+ GetVecinoMax(Mapa, Nodo, Mapa.IdCiudad[], float): Nodo
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Update(): void
«property»
+ nodoActual(): Nodo
Agent
Jugador
+ mapa: Mapa
- nodoActual: Nodo
- nodoObjetivo: Nodo
- parametros: EnvironmentParameters
- speed: float
+ target: GameObject
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
+ generaVirus(): void
+ getVecinoMax(Mapa, Nodo, Mapa.IdCiudad[], float): Nodo
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Update(): void
Agent
Jugador_Negativo
+ mapa: Mapa
- nodoActual: Nodo
- nodoObjetivo: Nodo
- parametros: EnvironmentParameters
- speed: float
+ target: GameObject
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
+ GetVecinoMax(Mapa, Nodo, Mapa.IdCiudad[], float): Nodo
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Update(): void
Agent
JugadorCura
+ mapa: Mapa
- nodoActual: Nodo
- parametros: EnvironmentParameters
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
FixedUpdate(): void
+ GeneraVirus(Mapa, Nodo, Mapa.IdCiudad[], float, float, List<Nodo>): void
+ GetVecinoMax(Mapa, Nodo, Mapa.IdCiudad[], float): Nodo
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Agent
JugadorDecidir
- nAcciones: int = 0
- nAccionesMaximas: int
+ nCuraciones: int = 0
+ nMovimientos: int = 0
- parametros: EnvironmentParameters
- speed: float
- tablero: Juego
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
FixedUpdate(): void
+ GeneraVirus(Mapa, Nodo, Mapa.IdCiudad[], float, float, List<Nodo>): void
+ GetVecinoMax(Mapa, Nodo, Mapa.IdCiudad[], float): Nodo
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Update(): void
«property»
+ nodoActual(): Nodo
Agent
JugadorDecidir2
- nAcciones: int = 0
- nAccionesMaximas: int
+ nCuraciones: int = 0
+ nMovimientos: int = 0
- parametros: EnvironmentParameters
- speed: float
- tablero: Juego
+ CollectDiscreteActionMasks(DiscreteActionMasker): void
+ CollectObservations(VectorSensor): void
FixedUpdate(): void
+ GeneraVirus(Mapa, Nodo, Mapa.IdCiudad[], float, float, List<Nodo>): void
+ GetVecinoMax(Mapa, Nodo, Mapa.IdCiudad[], float): Nodo
+ Initialize(): void
+ OnActionReceived(float[]): void
+ OnEpisodeBegin(): void
Update(): void
«property»
+ nodoActual(): Nodo

Bibliografía

[1] D. Silver, «Deep Reinforcement Learning,» 17 Junio 2016. [En línea]. Available: https://deepmind.com/blog/article/deep-reinforcement-learning.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra y M. Riedmiller, «Playing Atari with Deep Reinforcement Learning,» 2013.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel y D. Hassabis, «Mastering the game of Go with deep neural networks and tree search,» *Nature,* nº 529, p. 37, 2016.

[4] M. Pfeiffer, «Reinforcement Learning of Strategies for Settlers of Catan,» *Institute of Theoretical Computer Science,* p. 4, 2004.

[5] S. Arzt, G. Mitterlechner, M. Tatzgern y T. Stütz, «Deep Reinforcement Learning of an Agent in a Modern 3D Video Game,» p. 9, 2018.

[6] N. Baby y B. Goswami, «Implementing Artificial Intelligence Agent Within Connect 4 Using Unity3d and Machine Learning Concepts,» *International Journal of Recent Technology and Engineering,* vol. 7, p. 8, 2019.

[7] S. J. Russell y P. Norvig, Artificial Intelligence. A modern approach, Prentice Hall, 1995.

[8] B. Osiński y K. Budek, «What is reinforcement learning? The complete guide,» 5 Julio 2018. [En línea]. Available: https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/.

[9] A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar y D. Lange, «GitHub, Inc.,» 17 Noviembre 2019. [En línea]. Available: https://github.com/Unity-Technologies/ml-agents/tree/master/docs.

[10] R. Keim, «How to Train a Basic Perceptron Neural Network,» 24 Noviembre 2019. [En línea]. Available: https://www.allaboutcircuits.com/technical-articles/how-to-train-a-basic-perceptron-neural-network/.

[11] J. Dacombe, «An introduction to Artificial Neural Networks (with example),» 23 Octubre 2017. [En línea]. Available: https://medium.com/@jamesdacombe/an-introduction-to-artificial-neural-networks-with-example-ad459bb6941b.

[12] R. Parmar, «Training Deep Neural Networks,» 11 Septiembre 2018. [En línea]. Available: https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964.

[13] M. Silva, «Aprendizaje por Refuerzo: Introducción al mundo del RL,» 28 Abril 2019. [En línea]. Available: https://medium.com/aprendizaje-por-refuerzo-introducci%C3%B3n-al-mundo-del/aprendizaje-por-refuerzo-introducci%C3%B3n-al-mundo-del-rl-1fcfbaa1c87.

[14] R. S. Sutton y A. G. Barto, Reinforcement Learning An Introduction (Second Edition), Cambridge: The MIT Press, 2018.

[15] A. Karpathy, «Deep Reinforcement Learning: Pong from Pixels,» 31 Mayo 2016. [En línea]. Available: http://karpathy.github.io/2016/05/31/rl/.

[16] I. Asensio, «Qué es Unity y para qué sirve,» 8 Noviembre 2019. [En línea]. Available: https://www.masterd.es/blog/que-es-unity-3d-tutorial/.

[17] J. Schulman, O. Kilmov, F. Wolski, P. Dhariwal y A. Radford, «Proximal Policy Optimization,» 20 Julio 2017. [En línea]. Available: https://openai.com/blog/openai-baselines-ppo/.

[18] AurelianTactics, «Understanding PPO Plots in TensorBoard,» 14 Diciembre 2018. [En línea]. Available: https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2.

[19] Z-Man Games, «PANDEMIC,» [En línea]. Available: https://images-cdn.zmangames.com/us-east-1/filer_public/25/12/251252dd-1338-4f78-b90d-afe073c72363/zm7101_pandemic_rules.pdf.

[20] M. Merino, «Xataka,» 27 Enero 2019. [En línea]. Available: https://www.xataka.com/inteligencia-artificial/conceptos-inteligencia-artificial-que-aprendizaje-refuerzo.

[21] M. Merino, «Xataka,» 8 Noviembre 2018. [En línea]. Available: https://www.xataka.com/robotica-e-ia/ias-pueden-humillarnos-jugando-a-nuestro-videojuego-favorito-como-aprenden-a-hacerlo.

[22] C. Greene y N. Davis, «The power of Unity in AI,» 24 Julio 2020. [En línea]. Available: https://blogs.unity3d.com/2020/07/24/the-power-of-unity-in-ai/.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá